Managed Readiness Simulator (MARS) V2

Implementation of the Managed Readiness Model

Stephen Okazawa
Mike Ormrod
Chad Young

DRDC CORA TM 2010-261
December 2010

Defence R&D Canada
Centre for Operational Research & Analysis

Land Forces Operational Research Team

National Defence Défense nationale

Canada

Managed Readiness Simulator (MARS) V2

Implementation of the Managed Readiness Model

Stephen Okazawa
Mike Ormrod
Chad Young
DRDC CORA

Defence R&D Canada – CORA

Technical Memorandum
DRDC CORA TM 2010-261
December 2010

Principal Author

Original signed by Stephen Okazawa

Stephen Okazawa

Defence Scientist

Approved by

Original signed by Greg Smolynec

Greg Smolynec

Acting Section Head Land and Operational Commands

Approved for release by

Original signed by Paul Comeau

Paul Comeau

Chief Scientist DRDC CORA

Defence R&D Canada – Centre for Operational Research and Analysis (CORA)

Abstract

The Managed Readiness Simulator (MARS) is a versatile program that allows the user to quickly simulate a wide range of Canadian Forces readiness Scenarios to determine if the Resources of an Establishment are able to satisfy the requirements of a set of operational Tasks. The first version of MARS (V1) was successfully applied to a preliminary analysis of the Army's plans to generate the forces required for Task Force Afghanistan. However, several aspects of the MARS V1 architecture and design were identified as factors limiting the potential of MARS to address larger and more complex scenarios. After a significant redesign, the second version of MARS (V2) now incorporates a more advanced software architecture that integrates database technology into simulation execution and a new managed readiness model with a more advanced feature set that includes support for Establishment dynamics. The purpose of this paper is to document the implementation of the MARS V2 managed readiness model in the new software architecture. The intended audience is the analyst responsible for the implementation of MARS features and capabilities. The contents are both comprehensive and detailed such that the implementation of all aspects of the model can be understood and modified if necessary.

Résumé

Le programme de simulation de gestion de la disponibilité opérationnelle (programme MARS) est un programme polyvalent qui permet à l'utilisateur de rapidement simuler une vaste gamme de scénarios de disponibilité opérationnelle des Forces canadiennes afin de déterminer si les ressources d'un établissement sont en mesure de répondre aux besoins propres à un ensemble de tâches opérationnelles. La première version du programme MARS (V1) a été utilisée avec succès lors d'une analyse préliminaire des plans de l'Armée visant à mettre sur pied les forces nécessaires pour constituer la Force opérationnelle Afghanistan. Toutefois, plusieurs aspects de la conception et de l'architecture du programme MARS V1 ont été identifiés comme étant des facteurs limitant la capacité de MARS à traiter des scénarios plus importants et plus complexes. Après une restructuration en profondeur, la deuxième version de MARS (V2) incorpore désormais une architecture logicielle plus sophistiquée qui intègre une technologie de traitement de bases de données dans l'exécution de la simulation, ainsi qu'un nouveau modèle de gestion de la disponibilité opérationnelle possédant un ensemble de caractéristiques plus sophistiquées qui comprend du soutien au niveau de la dynamique de l'établissement. Le but de la présente étude est de documenter la mise en œuvre du modèle de gestion de la disponibilité opérationnelle du MARS V2 dans la nouvelle architecture logicielle. La public cible de ce document est l'analyste chargé de la mise en œuvre des caractéristiques et des capacités du programme MARS. Les points abordés sont tour à tour présentés de façon générale et en détail, de façon à ce que la mise en œuvre de tous les aspects du modèle soit bien comprise et modifiée au besoin.

This page intentionally left blank.

Executive summary

Managed Readiness Simulator (MARS) V2: Implementation of the Managed Readiness Model

Stephen Okazawa; Mike Ormrod; Chad Young; DRDC CORA TM 2010-261; Defence R&D Canada – CORA; December 2010.

Background: The Managed Readiness Simulator (MARS) is a software application being developed at Defence Research & Development Canada - Center for Operational Research & Analysis as an Applied Research Project managed by the Land Force Operational Research Team. MARS is designed to quickly simulate a wide range of readiness scenarios to determine if the resources of an Establishment are able to satisfy the requirements of planned operations.

The first version of MARS (termed V1) successfully conducted a preliminary analysis of the Army's plans to generate forces for Task Force Afghanistan. In the process of conducting this analysis, several aspects of the MARS V1 architecture and design were identified as limiting the potential of MARS to address future problems. In particular, modelling the dynamics of the Establishment (the creation, advancement and release of Establishment Resources), was not considered feasible within the MARS V1 software architecture and feature set. Certain design aspects of MARS V1 were also restrictive in terms of the types and complexity of scenarios that could be represented.

Results: Prior research produced a new technical platform for MARS (termed V2) which greatly expanded the potential of MARS to address more complex scenarios. This platform, called the Simulation Runtime Database (SRDB) approach, was exploited to develop more advanced modelling capabilities including Establishment dynamics. This involved the full reimplementation of MARS using the techniques and capabilities available in the new platform. Therefore, the purpose of this paper is to provide comprehensive and detailed documentation of the implementation of MARS V2.

Significance: This paper serves as the principal reference document describing exactly how MARS V2 executes simulations. It provides sufficient implementation detail on all aspects of the MARS model that analysts responsible for its future development and application will be able to understand it at the lowest level and modify it as necessary. This will allow MARS to continue to evolve to meet the increasingly complex readiness scenarios that help to inform military decision making.

Sommaire

Managed Readiness Simulator (MARS) V2: Implementation of the Managed Readiness Model

Stephen Okazawa; Mike Ormrod; Chad Young; DRDC CORA TM 2010-261; R et D pour la défense Canada – CARO; Decembre 2010.

Contexte : Le programme de simulation de gestion de la disponibilité opérationnelle (MARS) est une application logicielle en cours de développement au Centre d'analyse et de recherche opérationnelle de Recherche et développement pour la défense Canada. Il s'agit d'un projet de Recherche appliquée géré par l'Équipe de recherche opérationnelle de la Force terrestre. Le programme MARS est conçu pour rapidement simuler une vaste gamme de scénarios de gestion de la disponibilité opérationnelle dans le but de déterminer si les ressources d'un établissement sont en mesure de répondre aux besoins des opérations planifiées.

La première version de MARS (nommée V1) a permis d'effectuer avec succès une analyse préliminaire des plans de l'Armée visant à mettre sur pied des forces pour la Force opérationnelle Afghanistan. Dans le cadre de cette analyse, plusieurs aspects de l'architecture et de la conception de MARS V1 ont été identifiés comme étant des facteurs limitant la capacité de MARS à traiter les problèmes à venir. En particulier, la modélisation de la dynamique de l'établissement (la création, le développement et la publication des ressources de l'établissement) n'était pas considérée possible avec l'architecture logicielle et l'ensemble des caractéristiques de MARS V1. Certains aspects de la conception de MARS V1 étaient également restrictifs en termes de types et de complexité de scénarios pouvant être représentés.

Résultats : Des recherches précédentes ont produit une nouvelle plateforme technique pour MARS (nommée V2) qui a grandement augmenté la capacité de MARS à traiter les scénarios plus complexes. Cette plateforme, appelée approche de la base de donnée d'exécution de simulation (Simulation Runtime Database (SRDB)), a été utilisée pour élaborer des capacités de modélisation plus sophistiquées, y compris la dynamique de l'établissement. Cela impliquait une nouvelle mise en œuvre complète de MARS en utilisant les techniques et les capacités disponibles dans la nouvelle plateforme. En conséquence, le but de la présente étude est de fournir une documentation générale et détaillée de la mise en œuvre de MARS V2.

Portée : La présente étude doit servir de principal document de référence où est décrit avec exactitude de quelle façon MARS V2 exécute les simulations. Elle contient suffisamment de détails sur la mise en œuvre de tous les aspects du modèle MARS pour permettre aux analystes chargés de son développement et de son application futurs de comprendre le programme jusqu'à son niveau le plus élémentaire et à le modifier au besoin. Ceci permettra au programme MARS de continuer à évoluer et ainsi de répondre aux besoins de plus en plus complexes des scénarios de disponibilité opérationnelle et ce, dans le but d'aider la prise de décisions militaires.

Table of contents

List of figures

List of tables

1 Introduction

The Managed Readiness Simulator (MARS) is a versatile program that allows the user to quickly simulate a wide range of Canadian Forces (CF) readiness Scenarios to determine if the Resources of an Establishment are able to satisfy the requirements of a set of operational Tasks. The flexibility of MARS allows diverse operational tasks to be defined as processes composed of activities that place specific resource demands on the Establishment. The software also provides a graphical user interface that facilitates the creation and execution of simulation scenarios and the analysis of simulation output. The output analyzer allows the user to view aggregated simulation results and drill down to view the status of specific tasks and units over time. This provides the user with a powerful tool to anticipate problems that may arise and to identify their cause(s). Ultimately, MARS is intended to be used as a decision support tool for senior commanders of the CF. It provides them with forecasts of the impact of proposed changes to lines of operation, the Establishment, the readiness plan, CF policy, and other factors that may affect the CF's ability to satisfy operational demands and to maintain the health of the Establishment. A more detailed description of the motivation behind the development of MARS and its potential applications can be found in [1].

MARS is being developed by Defence Research & Development Canada - Center for Operational Research & Analysis (DRDC CORA) as an Applied Research Project managed by the Land Force Operational Research Team (LFORT). The first version of MARS (termed MARS V1) was an initial prototype developed in Rockwell's Arena simulation software [2] to demonstrate the managed readiness modeling concept. Previous publications [3][4], document the design and implementation of MARS V1. Completed in 2007, MARS V1 was successfully applied to a preliminary analysis of the Army's plans to generate the forces required for Task Force Afghanistan [5].

However, in the process of conducting this analysis, several aspects of the MARS V1 architecture and design were identified as factors limiting the potential of MARS to address future problems. In particular, the scale of simulation scenarios was limited by the MARS V1 Arena-based architecture and the implementation of important model features, such as Establishment dynamics (the creation, advancement and release of Establishment Resources), was not considered feasible. Therefore, the development of a new version of MARS (termed V2) was undertaken to build support for these capabilities. MARS V2 now incorporates a more advanced software architecture that integrates database technology into simulation execution [6] and a new managed readiness model with more advanced feature set that includes support for Establishment dynamics [7].

The purpose of this paper is to document the implementation of the MARS V2 managed readiness model in the new software architecture. The intended audience is the analyst responsible for the implementation of MARS simulation capabilities. This paper will provide sufficient detail on the implementation of MARS that all aspects of the model can be understood at the lowest level and modified if necessary. Users of MARS may also refer to this paper to gain an understanding of exactly how specific aspects of the model are implemented. In most cases, this paper will not discus the rational for or examples of the use of specific model features. These topics have been comprehensively addressed in prior publications [1][7].

The following two sections, 2 and 3, provide a review of MARS concepts, terminology and components. Section 4 provides an overview of the MARS V2 software architecture. Section 5 provides an overview of the MARS simulation lifecycle. Section 6 describes the data that underlies a MARS simulation scenario. Section 7 details the implementation of all parts of the MARS managed readiness model organized chronologically following the simulation lifecycle. Section 8 provides concluding remarks.

2 MARS Concepts and Terminology

The MARS program is designed to simulate a given readiness *Scenario* by forecasting the ability of an *Establishment* to generate the *Resources* required to satisfy a set of *Tasks* occurring over time under a given set of conditions. The program also records the state of every Resource throughout the simulation; therefore the results can be used to determine the utilization level of a unit within the Establishment or of a specific group of Resources.

The program currently models three types of Resources: Personnel, Equipment and Facilities. Every Resource occupies a *Slot* in an *Organization* as shown in Figure 1. In general, the *Attributes* of each Slot define a particular *Resource Requirement* of the Organization and the Slot can only be occupied by a single Resource that satisfies the requirement. For example a Slot may assert that it can only be filled by a Resource that satisfies the criteria: *rank == Captain AND occupation == Infantry*. However, special *Overflow* Slots can also be built into the Establishment that are allowed to contain many Resources. These Slots are sometimes needed to store extra Resources that do not satisfy the Resource Requirement of any Slot or that cannot be assigned to a Slot because eligible Slots are currently occupied.

An Organization consists of a group of Slots that define the Resource Requirements of a unit. They are typically arranged in a hierarchical tree structure where only the terminal nodes of the tree contain Slots. There are two types of Organizations. Establishment Organizations define the units that contain the Resources available in the Scenario. Theatre Organizations define templates of the units required by the Tasks being modelled and do not contain any Resources. During the simulation, the program uses these templates to create a group of Slots that will be filled by Establishment Resources selected to perform a Task.

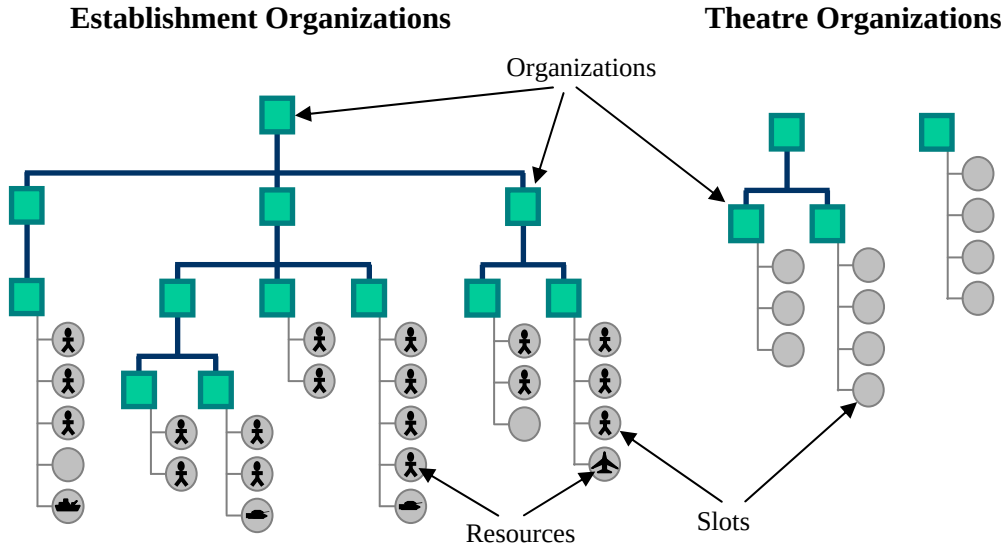

Figure 1: Establishment Organizations and Theatre Organizations showing Slots and Resources

Each Resource has Attributes that define its current state. Attributes store the information that determines whether a Resource can be chosen for a particular Task. Examples of Resource Attributes include a person's rank and qualifications. Other Attributes indicate whether the Resource is currently busy and whether there are any restrictions on what the Resource is allowed to do. In general, Attributes define the capability and availability of a Resource. These Attributes along with the Organization to which the Resource is attached are used to determine the eligibility of the Resource to be used by a given Task.

The operations and events being simulated in a MARS Scenario are represented by *Tasks* within the model. Each Task is broken down into *Activities* which are scheduled within the Task so that they will occur in a specified order. Activities are responsible for assembling the Resources they require. Activities can also be linked together to pass Resources from one Activity to another or to enforce dependencies. There are two types of Activities in the model:

1. A Process Activity temporarily employs Resources for a certain period of time and may alter their state upon commencement and completion.

2. An Event Activity changes the state of the selected Resources at a single point in time.

Each Activity is triggered in the simulation according to timing and Resource constraints. Activities simulate everything from training and operational Tasks to recruitment and retirement events or, if referring to equipment, acquisition and disposal events. The Activity construct is illustrated below in Figure 2.

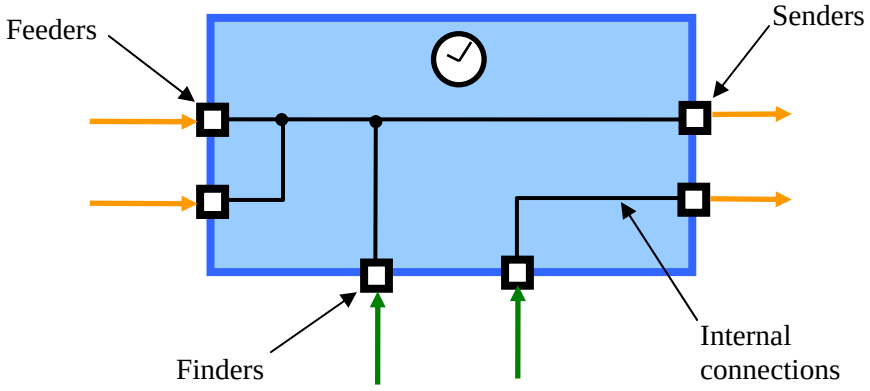

Figure 2: The Activity Construct showing Feeders, Finders and Senders and internal connections.

When an Activity is triggered and starts processing, it must acquire the Resources it needs to carry out its function. Resources enter an Activity as part of a resource group (*ResGrp*) through either a *Feeder* or *Finder* node and exit through a *Sender* node. An Activity may have multiple Feeders, Finders and Senders, and must have at least one Feeder or one Finder in order to act on at least a single ResGrp. A ResGrp is a set of Resources and Slots where each Resource occupies a single Slot and each Slot is either empty or occupied by at most one Resource. ResGrps are created by Finders which select Resources from the Establishment to participate in the Activity. Figure 3 illustrates the steps carried out by the Finders.

First, the Finder identifies the Theatre Organizations that contain the Slots that define the *Resource Requirement* for the Activity. For example, a Disaster Assistance Response Team Triage Unit might contain Slots for a medical officer, a medical technician, and a nurse.

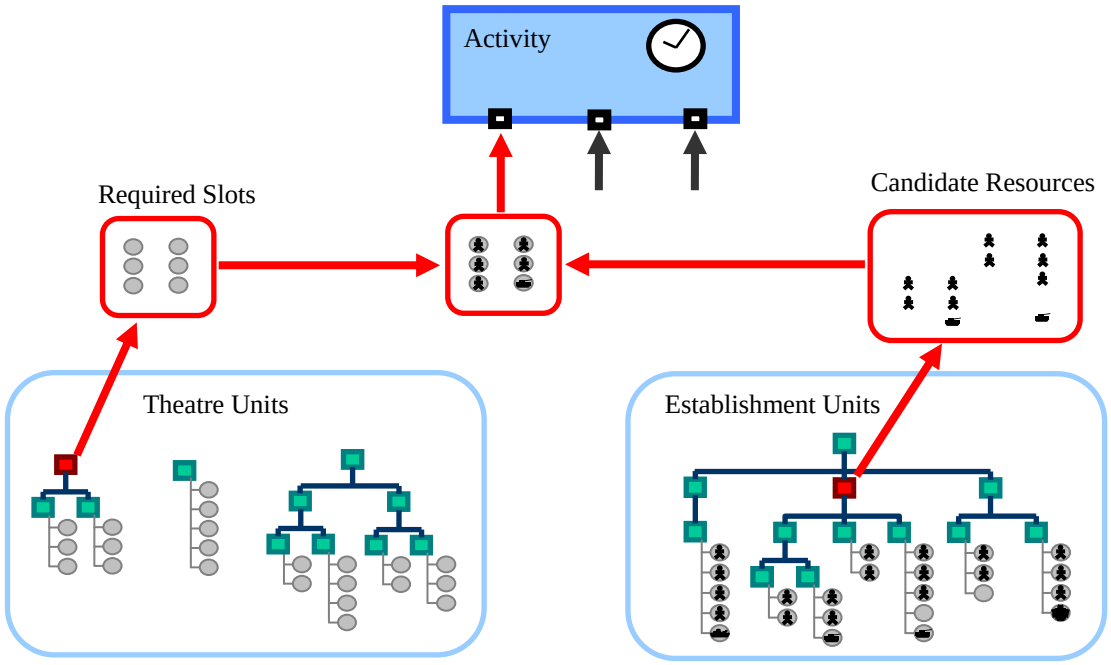

Figure 3: Resource selection process used by an Activity Finder.

Next, the Finder specifies a prioritized list of Establishment Organizations that may be searched to find Resources to participate in the Activity. This prioritized list is subject to constraints that can be used to limit the number of Resources taken from a given Organization and to filter for Resources with certain Attributes. The Finder also verifies that these Resources have not already been assigned to a conflicting Activity. This produces a list of *Candidate Resources*.

The Finder then attempts to fill each Required Slot with one of the Candidate Resources by comparing the Attributes of the Resources to the requirements of each Slot. If a suitable match is found, the Resource is assigned to the Slot and becomes part of the ResGrp being created by the Finder. To maximize the number of successful matches, the Finder attempts to assign the least qualified Candidate that meets the requirements of each Slot. For example, a Slot that can be filled by either a Major or Lieutenant Colonel will be preferentially assigned a Major because Lieutenant Colonels, being of higher rank, are in shorter supply and may be required by other Slots with more stringent requirements. This process is repeated for all Finders, with each Finder creating a ResGrp.

Activities also acquire Resources through Feeders which receive ResGrps that were created by a preceding Activity and passed on through one of that Activity's Senders. After acquiring its Resources through its Finders and Feeders, the Activity verifies that a specified minimum number of the required Slots have been filled. If this minimum requirement is not met, the Activity fails and the Resources are released. If sufficient Resources are found, the Activity takes control of

the selected Resources, altering their Attributes to reflect the nature of the Activity and employing them for the duration of the Activity.

Each Feeder and Finder is connected internally to a Sender. Upon Activity completion, each ResGrp is passed to a Sender which alters the Attributes of the Resources within the ResGrp to reflect the completion of the Activity. Each Sender is then responsible for either passing the ResGrps to the Feeder node of a follow-on Activity or for releasing the Resources within the ResGrp back to their Establishment unit. Senders and Feeders are the connection nodes that allow Activities to be linked together within a larger Task. When the Activity's processing time has finished and each ResGrp has exited through a Sender, the Activity is complete. Similarly, when all of the Activities of a Task are finished, the Task is complete.

To allow Tasks to be reused within a Scenario and to control when they begin, *Task Generators* are used to assign a start time to a Task. Multiple instances of a Task can be generated on a *Rotation* schedule to model the repetition of a Task such as a cycle of deployments that make up a continuous operation. When all of the Task Generators have been processed, and all of their associated Tasks are finished, the MARS Scenario is complete and the simulation stops.

From the outputs generated by the MARS simulation, the extent to which the Establishment was able to supply the Resources required for all the Tasks being modelled can be measured. More specifically, the output is analyzed to determine how successful each Finder was in creating its ResGrp from the Establishment. By aggregating the results from the Finders, the user can determine how successful the Establishment was in generating the required Resources for each Activity, Task, Task Generator, and the entire Scenario. Similarly, the states of the Resources within the Establishment can be tracked over time. These results can be combined to plot the state over time of a selected group of Resources, a unit or group of units, or the entire Establishment.

MARS is a versatile tool with many potential applications. Its strengths are its generic constructs that allow users to quickly simulate virtually any force generation Scenario and its outputs that provide users with the ability to aggregate and drill down into the simulation results to identify the causes of a particular outcome. The ability to forecast how successfully an Establishment can generate the forces required to satisfy both operational and sustainment demands will provide decision makers with invaluable information that currently is unavailable or very difficult to ascertain.

3 MARS Application Components

The MARS application consists of three major components, shown in Figure 4: a Scenario Database, a Managed Readiness Model, and a Graphical User Interface (GUI). The Scenario Database stores all the data that define a specific Scenario. The Managed Readiness Model is the discrete event representation of the process of selecting and employing Resources. The GUI allows the user to interact with the database and the model by performing three management functions. As the Input Manager, it is responsible for facilitating the transfer of Scenario data into the Scenario database. The input data consist of the Tasks, the Establishment, plans, and policies to be modelled in the proposed Scenario. The data may be input directly through the GUI input screens or imported from an external source such as a Corporate database or spreadsheet. As the Simulation Manager, the GUI controls the execution of the Simulation Scenario. Finally, as the Output Manager, it allows a user to analyze the simulation results and to generate output reports.

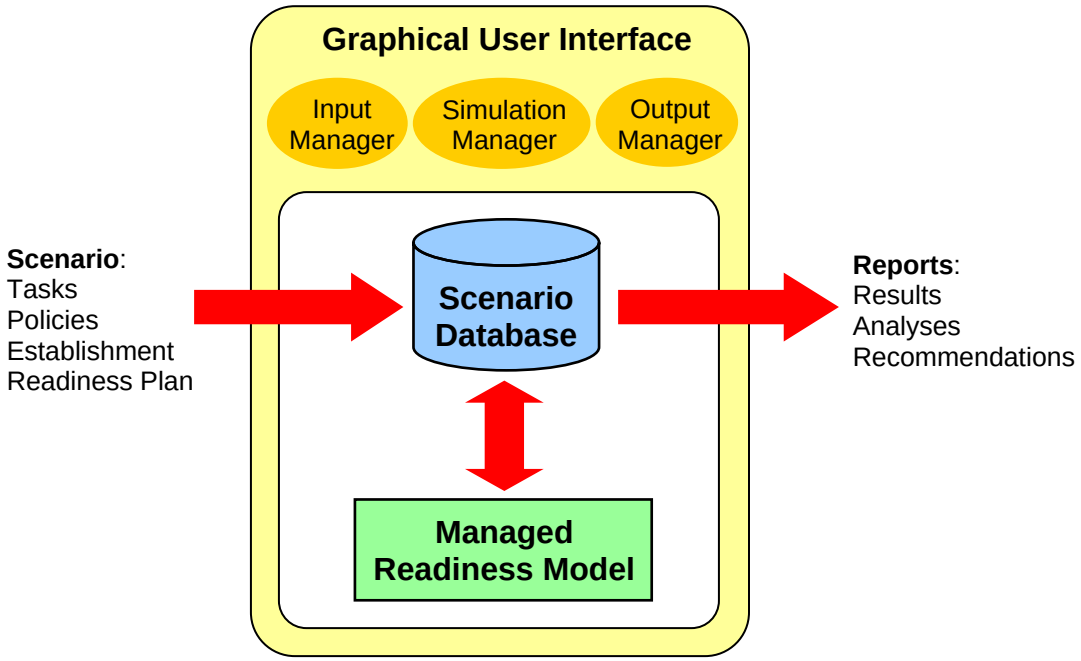

Figure 4: Components of the MARS Application with data transfer shown as red arrows.

4 Architecture Overview

The software architecture developed for MARS V2 is referred to as a Simulation Runtime Database (SRDB). In the SRDB approach, the simulation runtime data is stored in a relational database on the hard disk. This differs from traditional simulation methods that transfer input data from hard disk to memory for runtime processing and then back to hard disk for post processing. The advantage of keeping runtime data in a database is that the powerful data processing features provided by databases, including Structured Query Language (SQL), can be used within the simulation. This is particularly advantageous for large-scale resource management simulations that must perform complex operations on very large datasets. In MARS, the central process of selecting Slots required by an Activity, identifying candidate Resources, and matching the candidates to the Slots is efficiently implemented with SQL commands that interact with the database. The SRDB architecture is described in detail in [6].

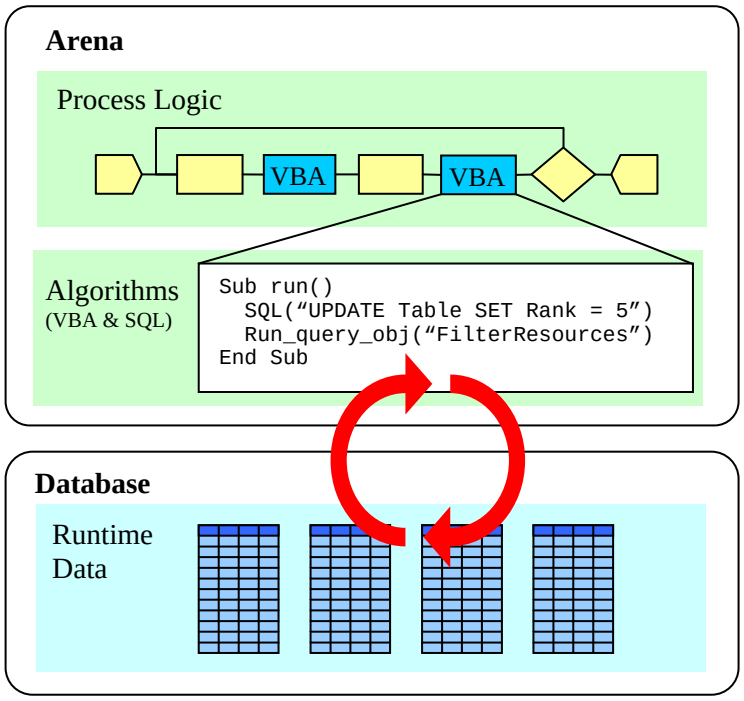

Figure 5: MARS V2 simulation architecture.

The MARS managed readiness model can be divided into the three layers shown in Figure 5. The top layer is the managed readiness process logic, implemented in Arena, which is the discrete event representation of the process of selecting and employing Resources for Activities. Below the process logic is an algorithm layer, also implemented in Arena, which defines the actions to be performed when events occur in the high level process. These algorithms are implemented in special Arena blocks that execute Visual Basic for Applications (VBA) code. The VBA code provides an interface between Arena and the MS Access database layer below it. Using VBA blocks, the algorithm layer can execute complex data operations on the database layer using SQL and can return information to the process logic, altering the course of the simulation.

MS Access, like many relational database packages, provides a graphical query design tool. This tool allows the simulation developer to build queries in a graphical interface that automatically generates the corresponding SQL syntax and saves the query as an object in the Access database. These saved SQL query objects can be invoked from code in the Arena VBA blocks. Figure 5 shows an example of running a saved query object called *Filter Resources*. In many cases, this is the preferred method of building and executing SQL queries for the MARS managed readiness model. This is because saved query objects are easier to create, edit and understand using the query design tool, and the SQL code is encapsulated in a named query object rather than being located somewhere in the model's VBA code. However, for straightforward data operations, the SQL command is typically defined as a string in the VBA code, such as the statement "*UPDATE Table SET Rank = 5*" shown in Figure 5. Which method to use is the choice of the simulation developer.

5 Simulation Lifecycle

This section provides an overview of the simulation lifecycle which is a high level representation of the MARS process logic. The detailed description of the implementation of the managed readiness model in Section 7 will proceed chronologically following the events in the simulation lifecycle.

Execution of a simulation Scenario in MARS consists of instantiating all the Activities that will be active in the Scenario and then running each Activity through a Discrete Event process representing the selection and employment of Resources. The entire simulation Lifecycle from simulation start to finish is illustrated in Figure 6. The simulation (shown in green) begins with the creation of Task generators. Each Task Generator (shown in blue) manages the creation of instances of a specified Task, also called Task Rotations. The Task Generator attaches timing information to each Rotation according to the specified Rotation schedule. Each Task Rotation (shown in purple) then instantiates the collection of Activities that constitutes its Task.

The full set of Activities from all Task Rotations is then placed in a holding queue. At this point, the simulation time starts advancing and the Activities wait in the queue until timing and other conditions are met that indicate that the activity is ready to launch.

When a Type 1 “Process” Activity (shown in solid red) is released from the queue, it processes its Feeders and Finders to assemble the Resources that will participate in the Activity. In so doing, it updates the Attributes of the acquired Resources, and keeps track of the number of Resources that it has acquired of each Type. The Activity then checks whether the number and Type of Resources found satisfies the Activity’s Firing Rules. If the Firing Rules are not satisfied, the Activity is cancelled and its ResGrps are immediately routed internally to Senders reflecting the Activity’s failed state. If the Firing Rules are satisfied, the Activity starts running, and employs its chosen Resources for the duration of the Activity. When this duration has passed, the Activity routes its ResGrps to Senders reflecting successful Activity completion.

Whether the Process Activity succeeded or failed, its ResGrps are processed at its Senders which perform Attribute updates and then either route the ResGrps onto subsequent Activities or return the Resources to the Establishment. This completes the lifecycle of a Process Activity, and the Task Rotation to which the Activity belongs registers its completion and whether it succeeded or failed.

When a Type 2 “Event” Activity (shown in dashed red) is released from the queue, it begins by processing its Finders. Event Activities have zero-duration and do not connect to other Activities so there are no Feeders or Senders. Event Activities also process Resources directly without matching them to a set of required Slots. Therefore, Activity Firing Rules are not enforced for Event Activities. Once the Finders have been processed and the Attributes of the selected Resources have been updated, the Event Activity has completed its life cycle. Its parent Task Rotation then registers its completion.

When all the Activities in a Rotation have completed, the Rotation is complete. Once all the Rotations of a Task have completed, the Task Generator is complete. Once all Task Generators have completed, the simulation is complete.

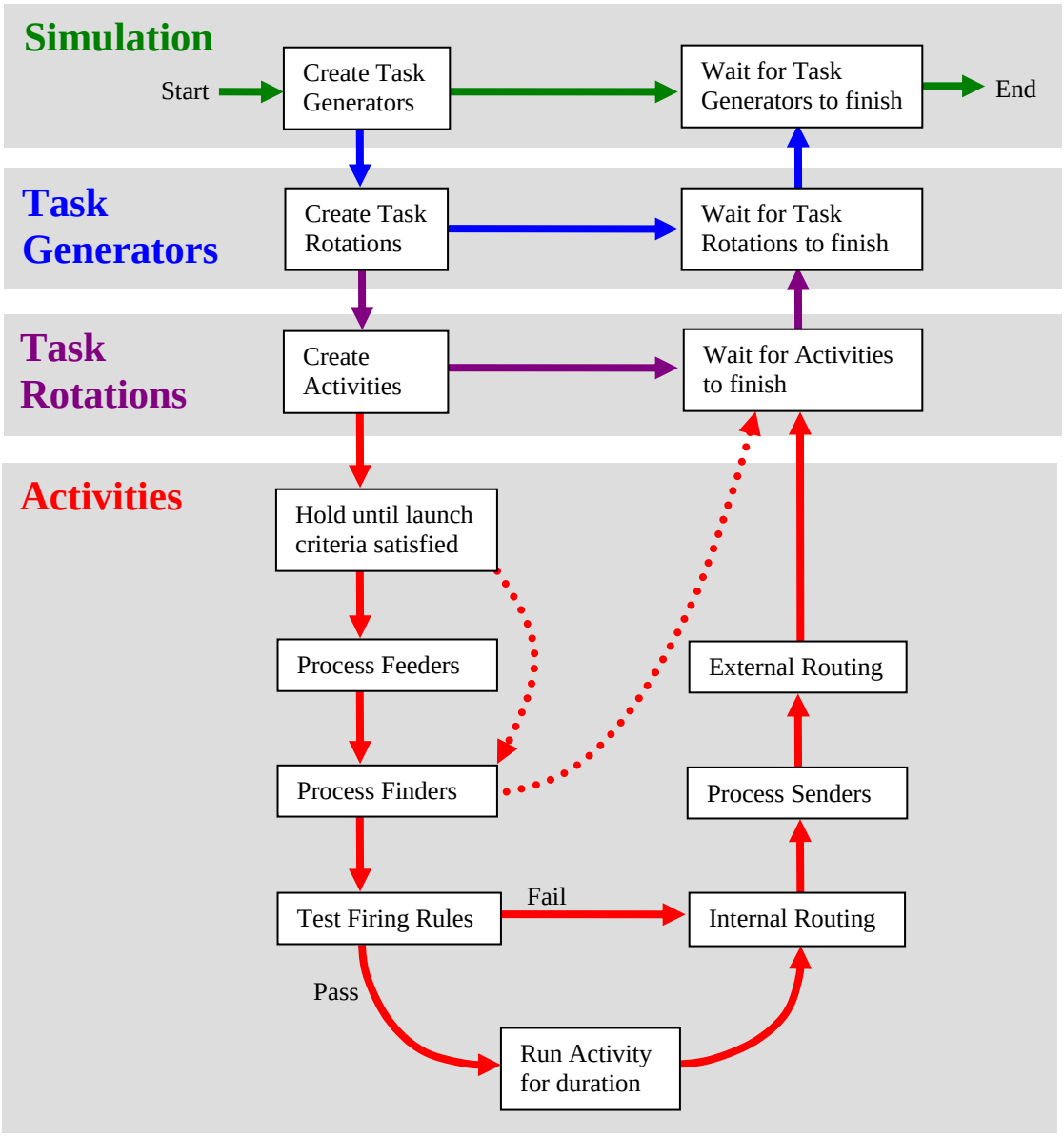

Figure 6: MARS V2 simulation lifecycle with the path for Type 1 (Process) Activities shown in solid red and the alternate path for Type 2 (Event) Activities shown in dashed red.

6 Simulation Runtime Data

In the SRDB approach, the entire simulation runtime dataset is stored in tables in a relational database. These data are queried and manipulated during execution using SQL commands embedded within the simulation process logic. Each of the tables in the database serves one of the roles described below in Table 1. For a complete list of tables and which role they perform, refer to Annex A.

Table 1: Description of roles played by MARS Runtime Data tables.

Table Role	Description
Scenario definition	These tables describe the simulation scenario to be executed. These data are created by the MARS user. They include the Establishment Organization, Slots, Resources, Tasks, Task Generators, Task Rotations, Activities, Feeders, Finders, Senders, Connections, and other data that represent the scenario the MARS user wishes to simulate. For dynamic objects such as Resources, these data contain their initial conditions.
Simulation state	These tables hold the current state of the simulation as the simulation runs. All objects that change over the course of the simulation (including Resources, Resource Attributes, ResGrps, Slot Attributes, and Activities) maintain their current state in a simulation state table.
Simulation record	These tables accumulate a record of simulation events. In coordination with updates to the simulation state tables, all objects that change over the course of the simulation record their state changes to a simulation record table. This allows the simulation state at any time during the run to be reconstructed based on the initial conditions and the simulation record tables. All simulation record tables have a *replication* field or are related to another table with a *replication* field so that simulation events from multiple replications in a Monte Carlo simulation can be recorded in the same database.
Intermediate results	These tables hold the intermediate results of an ongoing data operation. Intermediate result tables are used in a variety of situations, for example, to pass data to and from a function, to improve performance by storing frequently queried data, and to implement complex operations that cannot be completed in a single SQL query or nest of queries. Unlike the other three data roles above, intermediate results tables do not contain meaningful information outside the context of a particular runtime data operation.

Table Role	Description
Simulation output	These tables are used for generating outputs to analyse the outcome of the simulation scenario after execution. These data contain information from the user indicating parts of the simulation that the user wishes to investigate, and they store various result datasets involved in the requested analysis.
External data	These tables are involved in the transfer of data to and from external data sources.

Some tables contain both initial conditions (scenario definition data) and simulation record data. In these cases the initial conditions are identified with a zero in the table's *replication* field. All events recorded to the table during simulation execution will have an integer value greater than zero in this field indicating the replication number. When clearing these tables before the start of a run, only records with a non-zero value in the *replication* field are deleted to preserve the initial conditions.

7 Model Implementation

This section will describe the implementation of the MARS managed readiness model in Arena and MS Access. The Arena model is organized into interconnected sub-models, shown in Figure 7, reflecting the simulation lifecycle discussed above in Section 5. The following sub-sections will provide detailed descriptions of the process logic, algorithms and data tables involved in each sub-model. As a convention, the names of attributes, variables, subroutines, functions, database tables and fields are shown in italics. Subroutine and function names are followed by round brackets, e.g. *run()*, and table names are enclosed in square brackets, e.g. [*table*].

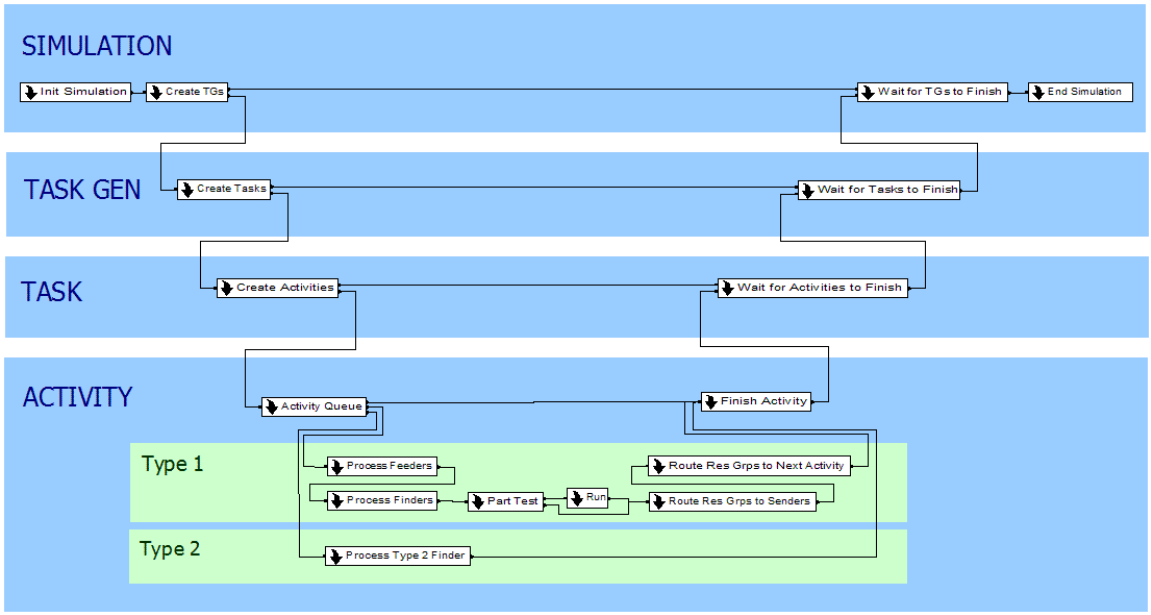

Figure 7: MARS V2 Arena model.

In implementing each of the sub-models, a number of conventions were followed to provide consistent visual documentation of the managed readiness model within the Arena file. These conventions were designed to improve the readability of the process logic making the model easier to modify and to enhance with new features. These conventions are listed in Table 2 and are illustrated in the sample sub-model shown in Figure 8.

Table 2: Description of MARS model file documentation conventions.

Documentation convention	**Description**
	A bold orange box encloses the discrete event logic inside each sub-model, visually separating the sub-model implementation from the inputs and outputs.
comment	Green comment text accompanies most Arena blocks providing a brief description of what is being done including the use of entity attributes whose scope is restricted to the current sub-model.
temp0 = n	Orange comment text indicates the assignment of a temporary entity attribute whose scope exceeds the current sub-model but whose use may change in different parts of the overall model. Temporary attribute names consist of the word "temp" followed by a number. For example, if one sub-model computes a value that is used in another sub-model, this information would be passed using a temporary entity attribute and would be commented in orange text.
save0 = m	Red comment text indicates the assignment of a permanent entity attribute. These are attributes that, once assigned, cannot be changed for the duration of the simulation. Permanent attribute names consist of the word "save" followed by a number. For example, an entity representing an activity will have a permanent attribute containing its unique ID number.
	Comments to the left and right of the enclosing orange box in a sub model indicate valid entity attributes entering and leaving the sub-model respectively. These comments are listed in either orange or red depending on whether the attributes are temporary or permanent. This allows the developer to keep track of which attributes are in use at the entry and exit of a sub-model. It also allows the developer to indicate when a temporary attribute is no longer in use by not listing it on the right (output) side of the enclosing box, meaning that it can be reused for another purpose in subsequent process logic.

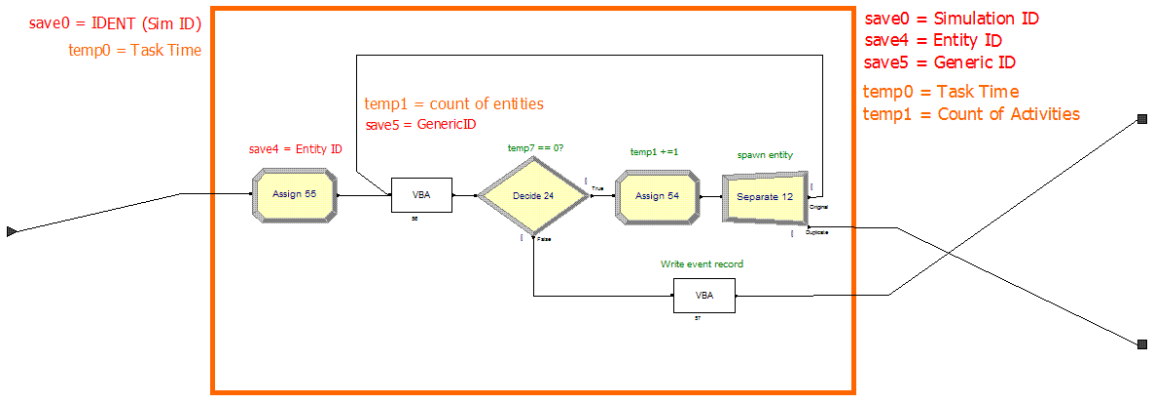

Figure 8: Sample sub-model showing documentation conventions.

7.1 Simulation Initialization

Before simulation execution commences, Arena automatically runs a user-defined VBA function called *ModelLogic_RunBeginSimulation()*. This function executes once before the first replication but not before subsequent replications. Due to the added complexity and as yet minimal benefit of running multiple replications, MARS has so far only been used to run single-replication simulations. However, the database and managed readiness model have been designed to support multiple replications so that Monte Carlo methods can be implemented in future analyses. The following list describes the sequence of actions performed in this initialization function.

1. Connect to the MS Access runtime database.
2. Set Arena's simulation end time attribute to the *RunEndTime* field in the [*Scenario Information*] table.
3. Set the number of simulation iterations to the *RunIterations* field in the *[Scenario Information]* table.
4. Store the *OutputSelectedOutcomes* field from the [*Scenario Information*] table to a global variable. This boolean variable sets the amount of information recorded during the process of matching Resources to Slots to either basic or detailed.
5. Write the current real date and time to the *RunDate* field of [*Scenario Information*]. This indicates to the MARS GUI that execution of the simulation scenario was initiated.
6. Initialize random number generation.
7. Determine the maximum *ResID* (resource ID) currently in use by querying the [*Resources*] table using the SQL statement show in Equation (1). Any new resources created during the simulation will be assigned *ResID* values that increment up from this initial maximum.

SELECT Max(Resources.ResID) AS MaxOfResID FROM Resources (1)

8. Set the first *AutoGenID* (automatically-generated resource ID) to 1. These are ID numbers given to virtual resources that do not exist in the Establishment and have no Attributes but which are automatically created to fill certain slots requiring a Resource type that is not being explicitly modelled in the scenario.

9. Call the *clear_sim_tables*() subroutine to clear all tables used to store simulation record information. Simulation record tables, shown in Table 3, contain simulation event data from all replications, thus they should only be cleared once before the first replication.

10. The initial condition of all Resources in the [*Resource Attributes*] definition table is copied to the [*OUTPUT_ResourceAttributes*] simulation record table, setting the *replication* field to 0.

Table 3: MARS Simulation record tables.

Table	**Role**
OUTPUT_ResourceAttributes	Record
Output_ResourceAutoGens	Record
ResGrp Progress	Record
ResGrp SearchPaths	Definition & Record
ResGrp SearchResults	Record
ResGrp SelectionResults	Definition & Record
Resources	Definition & Record
Simulation Progress	Record
Slot Attributes	Definition & Record

Once the *ModelLogic_RunBeginSimulation()* subroutine completes, the replications begin executing. All the actions performed by the simulation from this point on are repeated in each replication. The managed readiness model begins execution by creating a single entity at the create block in the *Init Simulation* sub-model, shown in Figure 9. This entity will be referred to as the Simulation Entity and is the highest-level Arena entity that begins and ends the simulation.

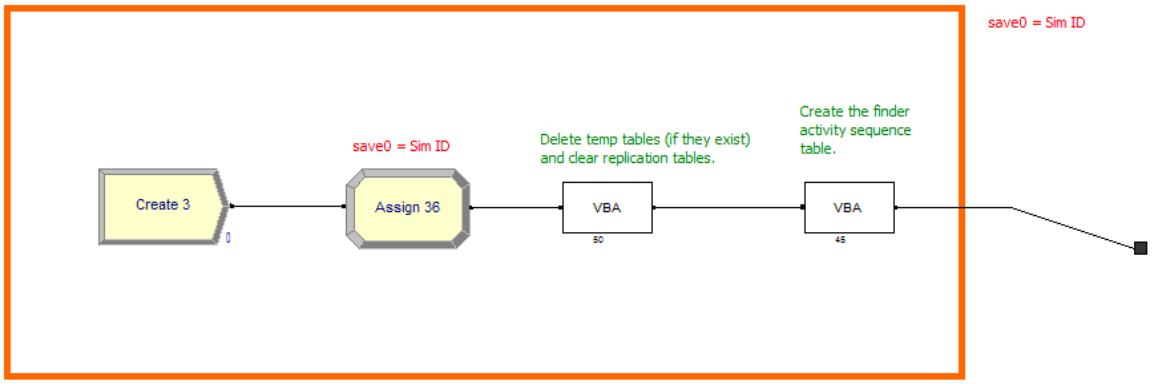

Figure 9: Init Simulation sub-model.

The Simulation Entity proceeds to an assign block that initializes to zero all the permanent attributes that will be used in the model, and it assigns a single *Sim ID* value to the *save0* attribute identifying the Simulation Entity. The *Sim ID* value is taken from Arena's internally-generated entity ID which is guaranteed to be unique for all entities.

A VBA block then records the first simulation event, *Simulation Start*, to the [*Simulation Progress*] record table by calling the *record_progress*() function. The [*Simulation Progress*] table records major events that track the progress of the simulation. Most sub-models record one or multiple events to this table providing information such as which Activity is executing, what action is being performed and what was the outcome. Each simulation progress record consists of a message ID (*SimProgressMsgID*), optional supporting data (*SimProgressValue*), the real date and time at which the record was created, and the current database file size. The [*Def SimProgressMsg*] table provides a list of available simulation progress messages as well as the meaning of optional supporting data if it is provided. For example, message 301 in this table is "Task Activities created" which includes supporting data indicating the number of Activities created. This [Simulation Progress] table is usually the primary source of information to identify what happened in a simulation scenario that failed or produced unexpected results. Calls to the *record_progress*() function will be mentioned in this document periodically but not necessarily thoroughly because they are evident in the VBA code.

The VBA code then clears all the simulation state and intermediate result tables, listed in Table 4, and then initializes certain simulation state tables to the initial conditions:

- The initial condition of all Resources in the [*Resource Attributes*] definition table is copied to the [*SO_ResAttrCurrent*] simulation state table.
- The initial condition of all Slots in the [*Slot Attributes*] definition table is copied to the [*SO_SlotAttrCurrent*] simulation state table.

Table 4: MARS Simulation state tables and intermediate result tables.

Table	**Role**
SO_AttrReqList	Intermediate
SO_AttrUpdate	Intermediate
SO_CloneSlotAttrReq	Intermediate
SO_FinderActSeq	Intermediate
SO_NewResAttrRecords	Intermediate
SO_PartTypeReqs	Intermediate
SO_RawResources	Intermediate
SO_RawSlots	Intermediate
SO_ReqSlots	Intermediate
SO_ResAttrCurrent	State
SO_ResGrpCurrent	State
SO_ResourceUpdate	Intermediate
SO_SelResources	Intermediate
SO_SelResourcesSave	Intermediate
SO_SlotAttrCurrent	State
SO_WeightedSum	Intermediate
Specific Activities	State

Table	Role
Specific Tasks	State
TEMP_RouteResGrptoSender	Intermediate
YC_ActiveActivities	Intermediate
YC_ActivityTree	Intermediate
YC_Check_ResSelectionRUL	Intermediate
YC_CombinedActiveFFS_SetAttr	Intermediate
YC_CombinedFFS	Intermediate
YC_CombinedFFS_SetAttr	Intermediate
YC_FinderActSeq	Intermediate
YC_ResGrpInitSpecActivity	Intermediate
YC_RUL_ShowRemovedResources	Intermediate

The code also initializes the intermediate result table [*YC_CombinedActive FFS_SetAttr*] by executing the stored query *YC_AP_CombinedFFS_SetAttr*. This query maps the sequence of Resource attribute updating instructions that are carried out as ResGrps pass through the Feeders, Finders and Senders of linked Activities. The resulting table is used in testing for Resource Utilization Level (RUL) conflicts during ResGrp creation which is described in Section 7.7.

A second VBA block calculates and stores Activity sequence information that is also used later as part of RUL conflict testing. This information is stored in the [*YC_FinderActSeq*] table. The code initiates a search starting at Activity Finders to discover the succession of Activities that follow it via internal and external Activity connections. The code repeatedly executes the stored query *YC_AP_FinderActSeq* which determines the next Activity in the sequence and appends it to the [*YC_FinderActSeq*] table. This process is repeated until no further Activity chains are discovered.

The Simulation Entity then proceeds to the Task Generator creation sub-model.

7.2 Task Generator Creation

In the Task Generator Creation sub-model, shown in Figure 10, the Simulation Entity spawns a new Task Generator entity for each active Task Generator specified in the simulation definition tables.

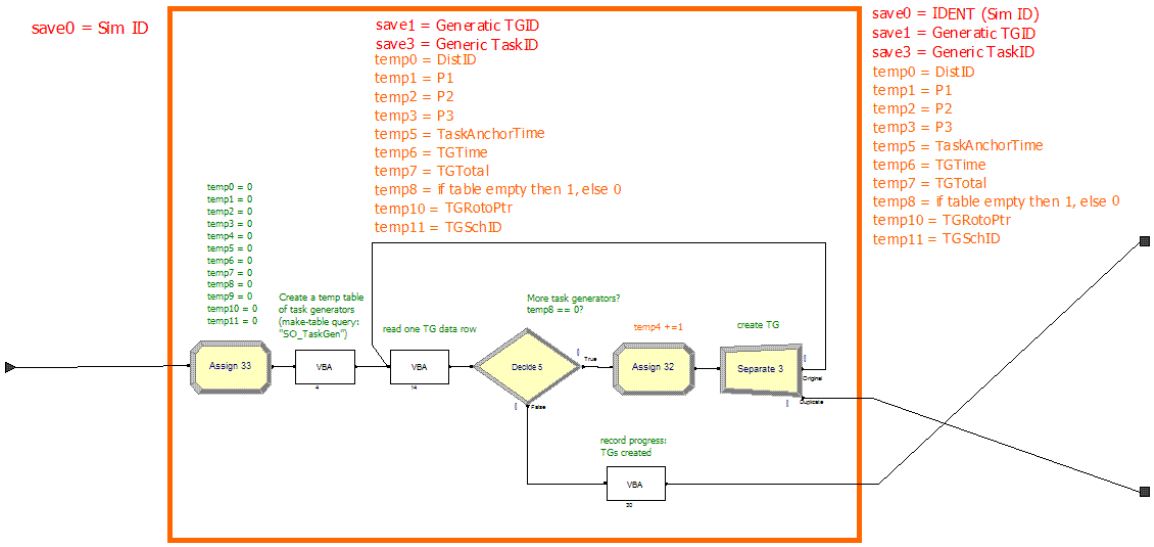

Figure 10: Task Generator creation sub-model.

The Simulation Entity first initializes to zero the temporary attributes that will be used throughout the model. A VBA block then executes a saved query called *SO_TaskGen* that creates a table called [*TEMP_TaskGen*] containing information on all the Task Generators that should be created. This query draws information from the [*Task Generator*] and [*TG Schedule*] tables. In the [*Task Generator*] table, the *Activate* field indicates which Task Generators should be created, the *TaskID* field indicates which Task type will be created, the *TGTotal* field indicates the number of Task Rotations that will be created and the *TGTime* field indicates when the first Task Rotation will run. The [*TG Schedule*] table indicates how the creation of the subsequent Task Rotations will be scheduled relative to the first Rotation.

The Simulation Entity then enters a loop that performs the following steps to create the new Task Generator Entities:

- Retrieve the next record (if it exists) from the [*TEMP_TaskGen*] table and save all the Task Generator information to permanent and temporary attributes. This information consists of parameters that determine how the Task Generator will later create Tasks which will be discussed in the next section.
- If no record could be retrieved in the previous step, then Task Generator creation is finished so exit the loop.
- Increment the count of created Task Generators stored in the *temp4* attribute.
- Spawn the new Task Generator Entity and loop back to the start.

Note that spawned entities inherit an identical set of attributes from the parent entity. The newly created Task Generator Entities proceed to the Create Tasks sub-model. When the Simulation Entity exits the loop, it passes through a VBA block that calls the *record_progress*() function to write a record to the [*Simulation Progress*] table indicating that Task Generator creation is complete and provides the number of Task Generators created. The Simulation Entity then

proceeds to the *Wait for Task Generators to Finish* sub-model where it is held until all created Task Generators complete processing.

7.3 Task Creation

In the *Create Tasks* sub-model, shown in Figure 11, each of the Task Generator Entities creates the specified number of Task Rotation Entities according to the Task Generator scheduling information. Each Task Rotation is an instance of Task type associated with the Task Generator. The specification of Task type, number of Task Rotations, and the schedule for their creation is contained in the Task Generator Entity attributes saved in the previous sub-model.

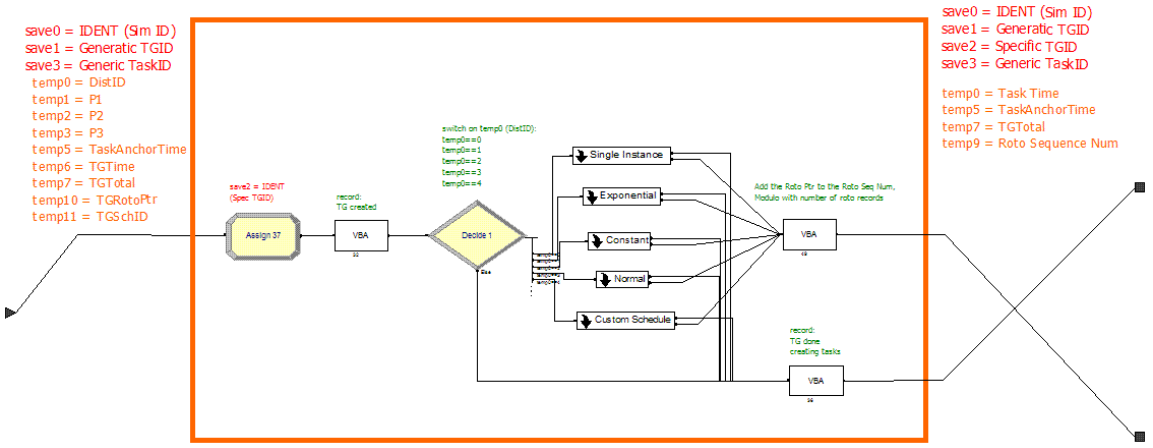

Figure 11: Create Tasks sub-model.

The Task Generator Entity is first assigned a unique ID called the *Specific TGID*. Then a VBA block calls the *record_progress*() function to write a message to the [*Simulation Progress*] table indicating that the Task Generator has been created. A decide block then determines which scheduling distribution type will be used. This information is stored in the *temp0* attribute. There are five distribution types which are defined in the [*Def Distributions*] table: single instance, exponential, constant, normal and custom schedule. The parameters *P1*, *P2*, *P3*, which are stored in attributes *temp1*, *temp2*, *temp3* respectively, determine the behaviour of the chosen distribution. Depending on the distribution type specified in *temp0*, the Task Generator Entity is directed to one of five sub-models that generate Task Rotations using the chosen schedule. Note that all Task Rotations (and the Activities that will be created in the next sub-model) are actually created at time zero. The scheduling information is stored in the Entity attributes that will eventually determine when Activity Entities will be launched later in the simulation.

In the single instance Task creation sub-model, shown in Figure 12, the Task Generator Entity spawns a single Task Rotation scheduled to run at the Task Generator's start time, *TGTime*. The created Task Entity then rearranges some of its temporary attributes as several attributes that carried scheduling information are no longer needed in subsequent model logic. Note that at the output from the sub-model the start time for the Task is now stored in the *temp0* attribute.

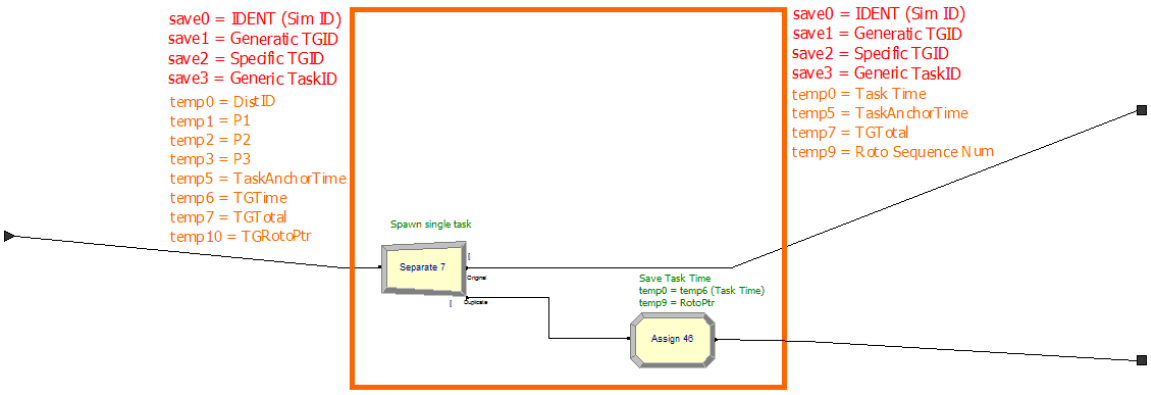

Figure 12: Single Instance Task creation sub-model.

In the exponential Task creation sub-model, shown in Figure 13, the Task Generator Entity creates the required number of Task Entities using an exponential distribution to determine the delay between the start times of subsequent Task Rotations. First, it sets up the *temp8* attribute to hold the Task start times, initializing it to the Task Generator start time, *TGTime*. It also zeroes the *temp9* attribute for use as a Task counter. The Task Generator Entity then enters a loop and creates the first Task Entity which receives the Task Generator's start time. It then increments the Task counter attribute and adds to the Task start time attribute using an exponential random number generator passing the *temp1* attribute as the exponential rate parameter. Finally, it checks whether the required number of Task Entities have been created. If so, the Task Generator Entity exits the loop; else, it repeats the loop spawning the next Task Entity using the new Task time. As newly created Task entities leave the loop, they store their given start time to the *temp0* attribute. The Task counter attribute, *temp9*, is inherited from the Task Generator Entity and functions as a Rotation sequence number for the Task Entity which is an important attribute used later during the processing of Finder nodes.

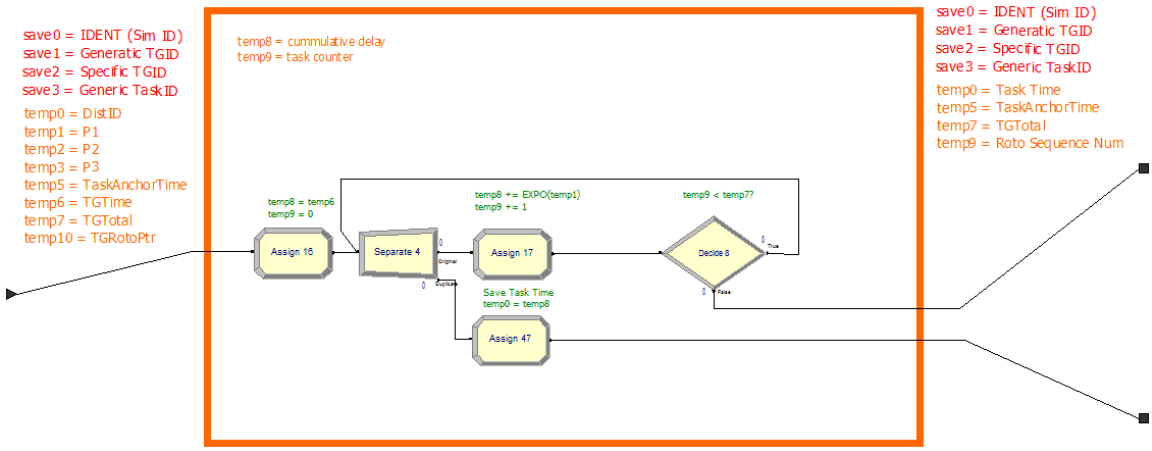

Figure 13: Exponential Task creation sub-model.

The constant and normal Task creation sub-models function in exactly the same manner as the exponential sub-model except for the distribution type used to determine the spacing between Task Rotation start times. For constant Task creation, the *temp1* attribute contains a constant

delay value between successive Task starts. For normal Task creation, the *temp1* and *temp2* attributes contain the mean and standard deviation values that specify the normal delay distribution between successive Task starts.

In the custom schedule Task creation sub-model, shown in Figure 14, the Task Generator Entity uses a user-defined schedule to set the times between successive Task Rotations. The Task Generator Entity first sets the *temp2* attribute to the start time for the Task Generator, *TGTime*, and sets the *temp9* attribute to zero to be used as the Task counter. It then enters a VBA block that executes the stored query *SO_TGSpacing*, passing in the ID of the Task Generator schedule to be used, *TGSchID*, as a parameter. This query creates a table named [*TEMP_TGSpacing*] containing the time delays to be applied between each successive Task Rotation. The Task Generator Entity then enters a loop which performs the following actions on each pass:

- Retrieve the next record from [*TEMP_TGSpacing*] in a VBA block, and store the delay time to the *temp1* attribute.
- If there were no records left in [*TEMP_TGSpacing*] then exit the loop.
- Add the delay time in *temp1* to the accumulated Task start time in *temp2*.
- Spawn the new Task Entity, which inherits the Task Generator Entity's current attributes.
- Increment the Task counter, *temp9*, and return to the start of the loop.

When the newly created Task Entity leaves the loop, it saves its start time to *temp0* and retains the Task counter attribute, *temp9*, as the Task Entity's Rotation sequence number.

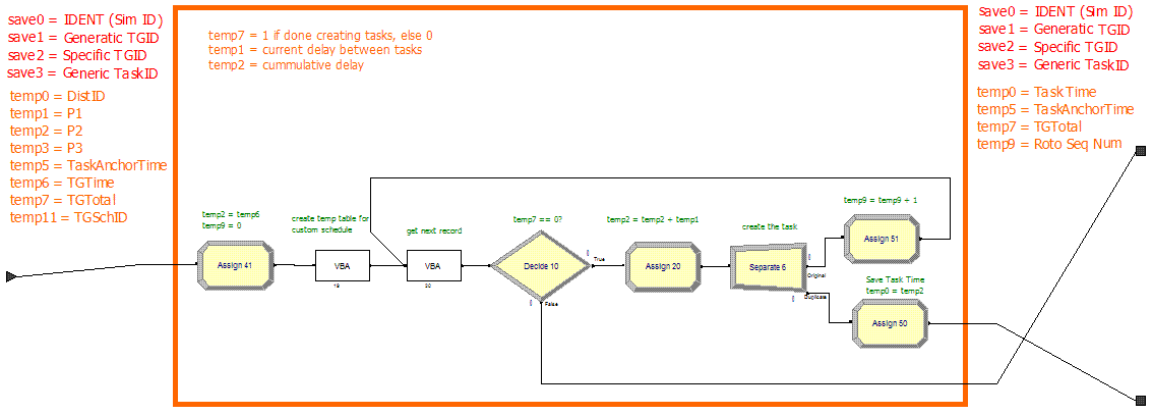

Figure 14: Custom Schedule Task creation sub-model.

Upon leaving the Task creation sub-models, the new Task Entities enter a VBA block in the Create Tasks sub-model in Figure 11. This VBA block uses the TGRotoPtr attribute stored in *temp10* to set the Rotation sequence number of the first Task which is stored in the *temp9* attribute. After the first Task, the Rotation sequence number increments by one up to the total number of Tasks and then wraps back to one and continues incrementing. For example, in a Task Generator that creates five Task Rotations, if the TGRotoPtr attribute was 3, the first chronological Task would be assigned Rotation sequence number 3, and subsequent Tasks would be assigned the sequence numbers 4,5,1 and 2. This allows the user to cause the first Task to use a

Rotation sequence number other than 1. The Task entities then proceed to the *Activity Creation* sub-model. After creating the specified number of Task Entities, the Task Generator Entities pass through a VBA block that calls the *record_progress*() function to write a record to the [*Simulation Progress*] table indicating that Task creation is complete and providing the number of Tasks created. The Task Generator Entity then proceeds to the *Wait for Tasks to Finish* sub-model where it is held until all created Tasks complete processing.

7.4 Activity Creation

In the *Create Activities* sub-model, shown in Figure 15, each Task Entity creates a collection of Activity Entities that make up the Task. Each Activity Entity is assigned a time window in simulation time units (days from simulation start) indicating when the Activity is allowed to start.

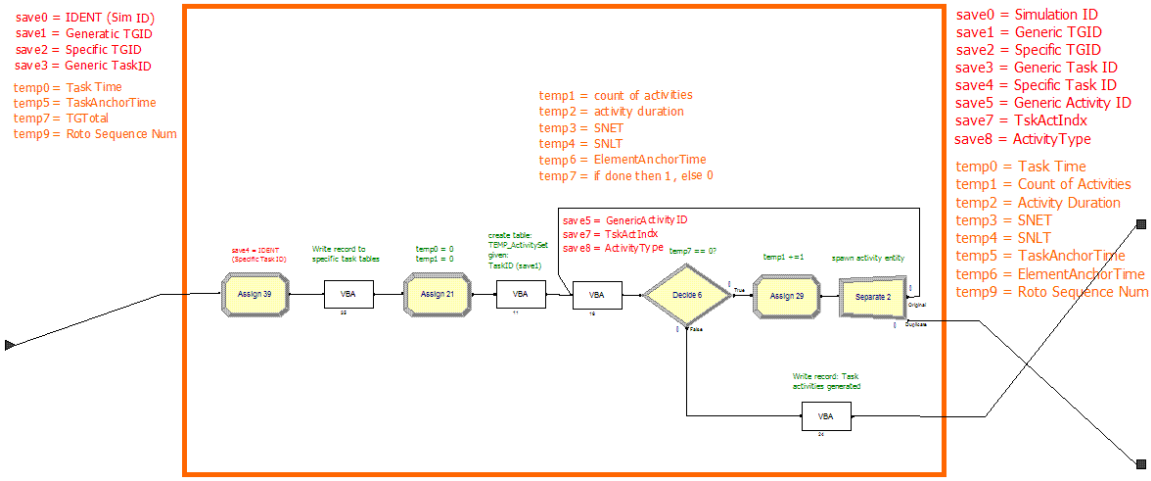

Figure 15: Create Activities sub-model.

The Task Entity is first assigned a unique ID value. In a VBA block, it then adds a record of itself to the [*Specific Tasks*] state table and records a "Task Created" message in the [*Simulation Progress*] table. It then initializes temporary attributes to begin the process of Activity creation.

To begin creating Activities, the Task Entity runs a stored query called *SO_ActivitySet* that gathers information from the [*Task Activity Sets*], [*Task Activity Timing*] and [*Generic Activities*] tables to create a temporary table called [*TEMP_ActivitySet*]. This table contains information on all the Activities that will be created. The Task Entity then enters a loop that performs the following steps:

- Retrieve the next record from the [*TEMP_ActivitySet*] table and save the Activity information to its Attributes.
- If no record was found in the previous step then exit the loop.
- Increment a counter of created activities stored in *temp1*.
- Spawn a new Entity to represent the Activity and return to the start of the loop.

The newly created Activity entities then proceed to the *Activity Queue* sub-model where they are held until their scheduled time arrives and specified conditions are met. When the Task Entity leaves the Activity creation loop, it calls the *record_progress*() function to write the message "Task Activities Created" to the [*Simulation Progress*] table including supporting data indicating the number of Activities created. It then proceeds to the *Wait for Activities to Finish* sub-model where it is held until all of the created Activities complete processing.

7.5 Activity Queue

In the Activity Queue sub-model, shown in Figure 16, the Activity Entities are held until they meet the following criteria that indicate that the Activity is ready to run:

- The simulation time is within the start time window for the queued Activity.
- All preceding Activities that are connected to the queued Activity have been completed and have routed their ResGrps through their Senders.
- All preceding Activities that the queued Activity depends on have been completed.

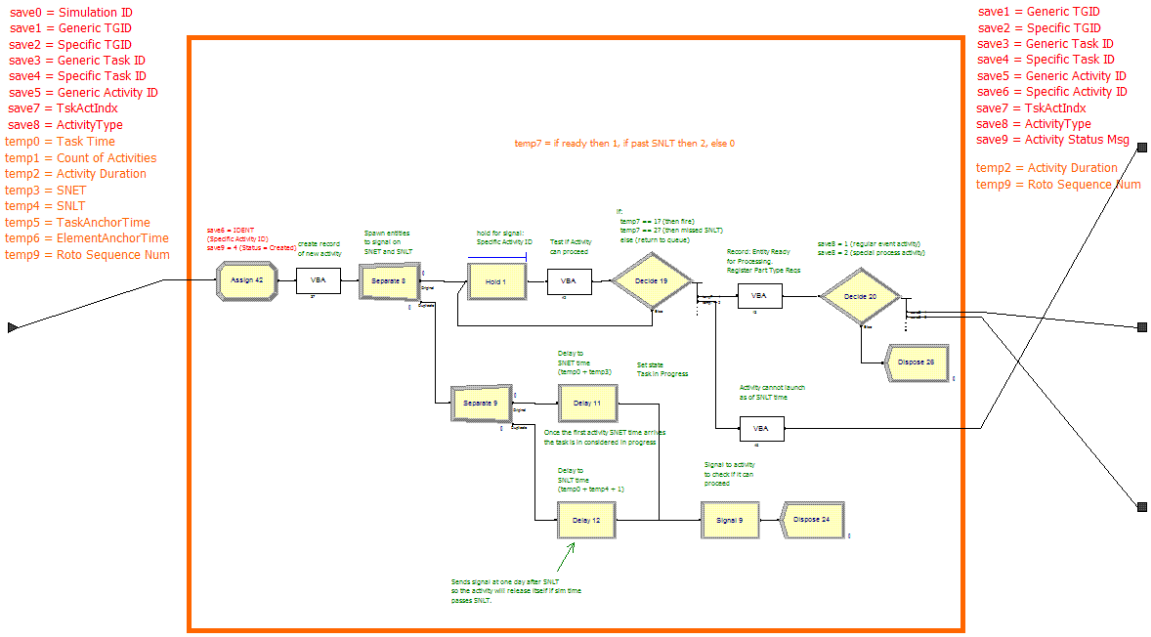

Figure 16: Activity Queue sub-model.

The mechanism that triggers the Activity Entities to test if they meet these criteria is based on signals. A signal is sent to a specific queued Activity when an event occurs that may mean the Activity is now ready to run. The Activity Entity responds to the signal by testing against the above criteria. If it passes, it exits the queue for processing. If it fails, it returns to the queue. Signals are sent to Activities in the holding queue in the following situations:

- The queued Activity's Start-No-Earlier-Than (SNET) time has arrived.
- A preceding Activity that is connected to the queued Activity has been completed.

- A preceding Activity that the queued Activity depends on has been completed.
- The queued Activity's Start-No-Later-Than (SNLT) time has passed.

A typical Activity that has connections and/or dependencies will receive multiple signals and test against the launch criteria multiple times before it satisfies all the launch criteria and exits the queue. The last signal above, indicating that the SNLT time has passed, cannot result in the Activity being released for regular processing (by definition), but allows the Activity to be removed from the queue and recorded as having failed to meet its launch criteria.

Starting at the entry to the sub-model, the Activity Entity first saves a Specific Activity ID to the *save6* attribute and an *Activity Created* status value to the *save9* attribute. In a VBA block, it then creates a record of itself in the [*Specific Activities*] status table and records an "Activity Created" message to the [*Simulation Progress*] table.

The code then executes the stored query *YC_AP_ResGrpInitSpecActivity* which adds records to the [*YC_ResGrpInitSpecActivity*] table to record the IDs of the ResGrps that the Activity's Finders will create. This table is used later during the creation of Resource Candidate lists.

The Activity Entity then spawns a new entity, which in turns spawns a second entity. These two spawned entities inherit all attribute values from the Activity Entity and are used to generate timing signals for the Activity. The first spawned entity waits in a delay block until the Activity's SNET time has arrived (stored in the *temp3* attribute). The second spawned entity waits in a delay block until one day after the SNLT time (stored in the *temp4* attribute). The original Activity Entity then enters the holding queue for all waiting Activities. Each Activity Entity waits in the queue for a signal whose value is the Entity's Specific Activity ID stored in the *save6* attribute. When either the SNET delayed entity or the SNLT delayed entity wakes up, it sends a signal with the value of the *save6* attribute. This signal is then received by the Activity Entity waiting in the queue. The Activity Entity will also receive a signal at the completion of any other Activity that is connected to its Feeders or that it depends on (these signals will be described in later sections). The Activity entity only responds to these signals while it is in the queue. Once, the Activity entity has left the queue, any signals sent to it have no effect.

When a queued Activity Entity receives a signal, it leaves the queue and enters a VBA block responsible for testing the Activity launch criteria. The result of the test is stored in the Activity Entity's *temp7* attribute. A result of 0 means the Activity has not yet met the launch criteria and should be returned to the queue. A result of 1 means the Activity has satisfied the launch criteria and should be released for processing. A result of 2 means the SNLT time has passed and the activity should be released from the queue and recorded as failing to meet its launch criteria before or at the SNLT.

Within the VBA block, the launch criteria test first checks if the simulation time is less than the Activity's SNET time. If so, then the *temp7* attribute is set to 0 and the Activity Entity exits the VBA block to be returned to the queue. Otherwise, the algorithm checks if the simulation time is less than the Activity's SNLT time meaning the simulation time is within the start time window for the Activity. If so, the code updates the Activity's state in the [*Specific Activities*] table to "Activity Ready". It then runs the query *SO_FeederResGrpArr2* which retrieves a list of Activity Feeder connections that have not yet received a ResGrp (indicating that a preceding connected Activity has not yet completed). If this query returns no records, the code runs a second query

SO_DependenciesMissing that retrieves a list of incomplete Activities that the Queued Activity depends on. If no records are returned from this query, then the Activity has met all the launch criteria and the *temp7* attribute is set to 1 indicating that the Activity can exit the holding queue for processing. If either of these queries does return records, then the *temp7* attribute is set to 0.

If the simulation time has passed the Activity's SNLT, the *temp7* attribute is set to 2. The code then attempts to determine why the Activity failed to launch within the start time window. First, it executes the query *SO_FeederResGrpArr2* to retrieve a list of Feeder connections that did not receive a ResGrp. If the query returns records, it saves the first Feeder index that did not receive a required ResGrp. The code then updates the Activity's state in the [*Specific Activities*] table to "Activity Failed (Waiting for Feeder ResGrp)," and it provides the Feeder index as the message detail. It also records a corresponding message to the [*Simulation Progress*] table. If the query returned no records, the code executes the query *SO_DependenciesMissing* to retrieve a list of incomplete Activities that the Queued Activity depends on. If the query returns records, it saves the first Activity ID. The code then updates the Activity's state in the [*Specific Activities*] table to "Activity Failed (Waiting for Dependent Activity)," and it provides the Activity ID as the message detail. It also records a corresponding message to the [*Simulation Progress*] table.

After passing through the launch criteria test VBA block, the Activity enters a decide block that routes the Activity depending on the value of its *temp7* attribute. If *temp7* is 0, the Activity Entity failed to meet the launch criteria but the SNLT time has not yet arrived so it is routed back to the holding queue.

If *temp7* is 1, the Activity met the launch criteria and is ready to continue to Resource selection. In this case, the Activity Entity enters another VBA block that first records the message "Activity Released from Queue" to the [*Simulation Progress*] table. The code then clears a table called [*SO_PartTypeReqs*] and then runs an append query *SO_RegFiringRules* on this table that inserts the number of resources of each type that must be found in order to successfully run the Activity. A decide block then separates Type 1 Activities from Type 2 Activities, based on the *save8* attribute, and routes them to the appropriate resource selection process logic.

If *temp7* is 2, the Activity failed to meet its launch criteria and the SNLT time has passed. In this case the Activity proceeds directly to the Finish Activity sub-model.

7.6 Process Feeders

In the Process Feeders sub-model, shown in Figure 17, Type 1 Activity Entities that have been released from the Activity Queue begin assembling the Resources that will participate in the Activity by processing ResGrps that have arrived at the Activity's Feeders.

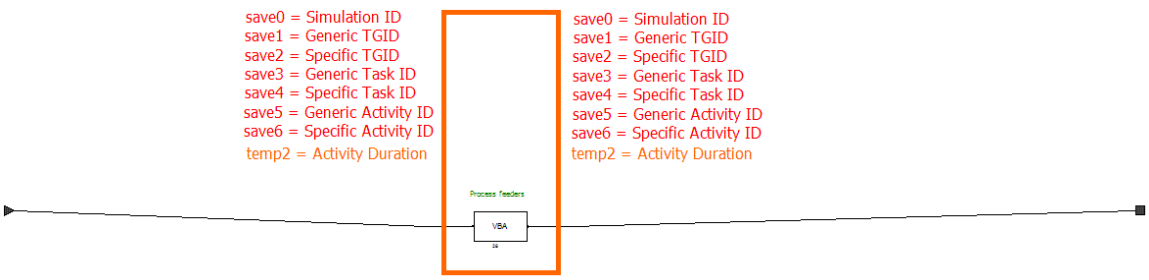

Figure 17: Process Feeders sub-model.

This sub-model consists of a single VBA block. The VBA code first runs the stored query *SO_FeederResGrp2* which retrieves a list of Feeders, the Resource Part Type associated with each Feeder and the ResGrps that have arrived at each Feeder. Resource Part Types are arbitrary ID numbers assigned by the user to Feeders and Finders in order to divide the Activity's total Resource requirement into parts. For example, in a training Activity, a Finder that assembles a ResGrp of students could be assigned Part Type 1 and another Finder that assembles a ResGrp of instructors could be assigned Part Type 2. For each Part Type, the Activity specifies a minimum percentage of Slots that must be filled.

For each ResGrp found by the query, the code determines the total number of Slots by calling the function *count_slots_in_grp()* and the number of Slots that are occupied by a Resource by calling the function *count_res_in_grp()*. These functions retrieve this information by executing the stored queries *SO_CountSlotsInGrp* and *SO_CountResInGrp* respectively. The code then updates the [*SO_PartTypeReqs*] table with this information using the following SQL query:

UPDATE SO_PartTypeReqs SET NumSupplied = NumSupplied + <ResCount>, NumTotal = NumTotal + <SlotCount> WHERE PartType = <iPartType> (2)

where <ResCount>, <SlotCount> and <iPartType> are parameters specifying the number of Resources, the number of Slots and the Part Type respectively.

For each Feeder, the code records a message "Activity Processing Feeder" to the [*Simulation Progress*] table providing the Feeder index. For each ResGrp it records a message "Activity Receiving Resource Grp" to the [*Simulation Progress*] table providing the ResGrp ID. The code then adds a record to the [*ResGrp Progress*] table indicating the simulation time, the Feeder index, and the Specific Activity ID at which the ResGrp was received by calling the function *record_res_grp_progress*().

The last step in the Feeder process is to update the attributes of the Resources in the current ResGrp. These attribute updates alter the state of the Resources to reflect the nature of the Activity. For example, an Activity that is part of an Expeditionary Operation will update the *Doing Status* attribute of incoming Resources to "On Exped OP." The following sub-section will describe the implementation of attribute updating which is also used by the Finder and Sender nodes. Once this step is complete, the code repeats this process for the next ResGrp that has arrived at an Activity Feeder.

Resource Attribute Updating

Attribute updating is performed on Resources as they start an Activity at Feeders and Finders and again as they leave the Activity through Senders. The update is performed in three steps:

1. Retrieve the list of Attribute updating instructions;

2. Identify the list of Resources whose attributes are to be updated; and

3. Apply the Attribute updating instructions to the identified Resources.

In the first step, implemented in the function *setup_attr_update*(), the VBA code first clears the intermediate table [*SO_AttrUpdate*] which stores the current attribute updating instructions. The code then executes one of three stored SQL queries depending on whether the current node is a Feeder, Finder or Sender: *SO_FeederAttrUpdate*, *SO_FinderAttrUpdate* or *SO_SenderAttrUpdate* respectively. Each of these queries takes the current Activity and Feeder, Finder or Sender index and populates the [*SO_AttrUpdate*] table with the corresponding attribute update instructions. Each row in the [*SO_AttrUpdate*] table is a single instruction consisting of an Attribute update type ID (*SetAttrTypeID*), an Attribute ID (*AttrID*), an Attribute value (*AttrValue*) and a Variable ID (*VariableID*). The typical behaviour is to set the Resource's Attribute to the given Attribute value, in which case the SetAttrTypeID field is set to "set equal to value." However, the SetAttrTypeID can also be set to "add value" or "subtract value" in order to increase or decrease the Resource's Attribute by the given Attribute value. In some cases, if the desired Attribute value is not a constant, the Variable ID can be used to specify a variable or computed value. During Attribute updating, the only Variable ID currently available is the current simulation time.

In the second step, implemented in the function *setup_resource_update*(), the VBA code clears the intermediate table [*SO_ResourceUpdate*] which contains a list of Resources whose attributes will be updated. For Type 1 (Process) Activities, the code executes the stored query *SO_AttrUpdate_ResGrp*, which appends the Resources contained in the currently processing ResGrp to the [*SO_ResourceUpdate*] table. For Type 2 (Event) Activities, ResGrps are not passed from Activity to Activity through Senders and Feeders, so Attribute updates only occur while processing Finders. Therefore, the code executes a different stored query *SO_AttrUpdate_ResGrpType2* which retrieves the list of Resources from another temporary table [*SO_SelResources*] and appends it to [*SO_ResourceUpdate*]. The [*SO_SelResources*] table is used during Finder processing to hold the list of Resources selected from the Establishment to participate in the Activity.

In the third step, implemented in the function *attribute_update*(), the VBA code performs a sequence of operations that carry out the Attribute update instructions on the identified Resources. The operations involved in this step depend on the type of update instructions to be performed. The code first checks if the update instructions include updates to the Resources' Slot Attribute. This is accomplished by executing the SQL query shown in Equation (3).

```
SELECT * FROM SO_AttrUpdate WHERE AttrID = 5
```

(3)

where AttrID 5 is the Resource Slot Attribute.

The Slot Attribute holds the ID of the Establishment Slot currently occupied by the Resource. Updates to this Attribute indicate that Resources are moving within the Establishment which involves additional operations during Attribute updating.

If there are updates to the Resources' Slot Attribute, the code executes queries that update the Resource Count Attribute of the affected Slots in the [*SO_SlotAttrCurrent*] state table and the [*Slot Attributes*] record table. First, the stored query *SO_AttrUpdate_SlotChangesTO* increments the current Resource Count attribute in [*SO_SlotAttrCurrent*] for Slots that are being moved into, and the stored query *SO_AttrUpdate_SlotChangesFROM* decrements the current Resource Count attribute for Slots that are being vacated. Then the stored queries *SO_AttrUpdate_SlotAttrRecINCR* and *SO_AttrUpdate_SlotAttrRecDECR* append records of increments and decrements to the Resource Count attribute for all affected Slots in the [*Slot Attributes*] table.

The code then performs updates on multi-item Attributes if they are present. These are Attributes that hold a list of items from a set of possible items. For example, personnel qualifications are stored in a multi-item Attribute because an individual may hold many qualifications at once. Rather than setting the value of this type of Attribute, the Attribute value specified in the update instruction is either added to or removed from the current list belonging to the Resource. The stored query *SO_AttrUpdate_MultiAppend* performs additions to multi-item Attributes by appending the appropriate records to the [*SO_ResAttrCurrent*] state table. To remove items from multi-item attributes, the stored query *SO_AttrUpdate_MultiRemove* retrieves a list of items to be removed from multi-item Attributes. The code then iterates through this list and performs the necessary deletions from the [*SO_ResAttrCurrent*] table.

The code then records these additions to and deletions from multi-item attributes in the [*OUTPUT_ResourceAttributes*] record table using the stored queries *SO_AttrUpdate_RecMultiAppend* and *SO_AttrUpdate_RecMultiRemove* respectively.

The [*OUTPUT_ResourceAttributes*] table has two time fields, *StartTime* and *EndTime*. The typical action while updating a Resource Attribute is to insert the current simulation time in the *StartTime* field when a record is inserted into the table. This is the case for additions to multi-item Attributes. However, deletions from multi-item attributes are distinguished by recording the current simulation time in the *EndTime* field. In this way, an item that is added to and later removed from a Resource's multi-item Attribute will have two records in the [*OUTPUT_ResourceAttributes*] record table: the first indicating when the item was added in the *StartTime* field, and the second indicating when the item was removed in the *EndTime* field.

The code then clears the temporary table [*SO_NewResAttrRecords*] in order to store the set of altered Attribute records that will later be appended to the [*OUTPUT_ResourceAttributes*] record table. In addition to recording new Attribute values, each record in the [*OUTPUT_ResourceAttributes*] table records the previous Attribute value and the current and previous Establishment Organization where the Resource is located. This additional data is recorded to facilitate post-processing for the generation of output reports. How this data is used in post-processing is beyond the scope of this paper.

If there are updates to the Slot Attribute, the entire set of Attributes for each Resource prior to the update (except for multi-item attributes which were dealt with separately above) is copied to the

[*SO_NewResAttrRecords*] table using the *PrevAttributeVal* field to store pre-update Attribute values. This is accomplished by executing the stored query *SO_AttrUpdate_RecAttrOldALL*. If there are no updates to the Slot attribute, only those Attributes that are to be updated are copied to the [*SO_NewResAttrRecords*] table using the *PrevAttributeVal* field by executing the stored query *SO_AttrUpdate_RecAttrOld*. The recording of the full set of Resource Attributes when there are Slot Attribute updates is performed to facilitate post-processing of Resource Attribute data aggregated by Establishment Organization (as noted before).

Finally, the Attribute update instructions (other than multi-item Attribute updates which were performed above) are applied to each identified Resource by executing the stored query *SO_AttrUpdate_SetAttr*. This updates the Attribute values in the [*SO_ResAttrCurrent*] state table.

To complete the Attribute update record, the new Attribute values are recorded to the [*SO_NewResAttrRecords*] table. If there are updates to the Slot Attribute, the entire set of Attributes for each Resource after the update is copied to the [*SO_NewResAttrRecords*] table using the *AttributeVal* field by executing the stored query *SO_AttrUpdate_RecAttrNewALL*. If there are no updates to the Slot Attribute, only those Attributes that were updated are copied to the [*SO_NewResAttrRecords*] table using the *AttributeVal* field by executing the stored query *SO_AttrUpdate_RecAttrNew*.

The last step in the *attribute_update()* function is to copy the contents of the [*SO_NewResAttrRecords*] intermediate table into the [*OUTPUT_ResourceAttributes*] record table using the stored query *SO_AttrUpdate_AppendNew*.

7.7 Process Finders

In the Process Finders sub-model, shown in Figure 18, Type 1 Activity Entities select Resources from the Establishment to be employed for the duration of the Activity. This involves a large number of complex operations that simulates the manner in which CF resources are assigned to Operations. All these steps are implemented in a single VBA block.

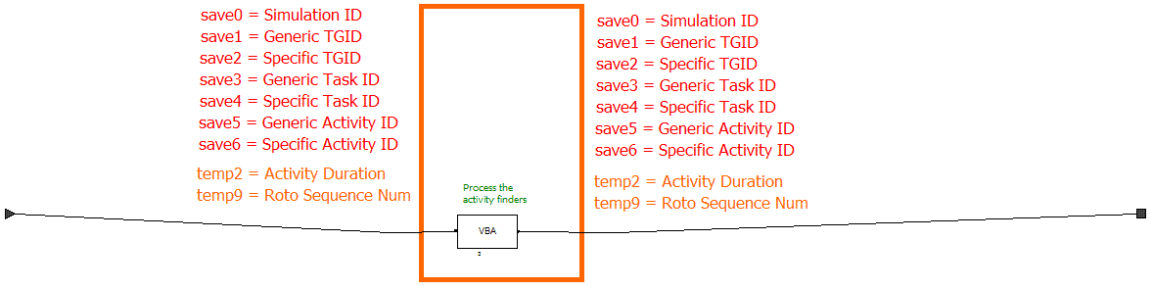

Figure 18: Process Finders sub-model.

The VBA code begins by retrieving information about each Finder node belonging to the Activity by executing the stored query *SO_ActivityFinders*. The code then processes each Finder in the list in order of the Finder index.

The first step in processing a Finder is to collect various parameters that define the behaviour of the Finder from the result set of the query executed above. These parameters will be used throughout the Process Finders sub-model and are listed below in Table 5.

Table 5: Description of Finder parameters.

Finder parameter	Description
FinderRecNum	The unique ID number for a specific Finder on a specific Activity.
FinderIndex	The Finder index number within the Activity.
TotalLimitTypeID	The type of limit to be applied to restrict the total number of Resource Candidates that may be used by the Finder to fill Slots required by the Activity.
TotalLimitValue	The specific limit value (for example, a percentage) used by the Total Limit Type.
SlotSortID	The type of Sorting that will be used to arrange the order in which Activity Slots will be filled.
ResourceSortID	The type of Sorting that will be used to arrange the order in which Resources will be matched to Slots.
FinderMatchID	The type of process that will be used to match Resources to Slots.
ResReqID	The identifier of the specific Resource requirement for the Finder.
PartType	The Part Type that the Resources selected by the Finder will be classified as.

The code then executes the stored query *SO_FinderResGrpID* which retrieves the *ResGrpID* and the *ResCanListID* for this specific Finder. The *ResGrpID* is the identifier for the ResGrp that will be created by the Finder. The *ResCanListID* is the identifier for the list of Establishment Organizations and ResGrps that will supply candidate resources to the Finder process.

The code then appends a record to the [*ResGrp Progress*] record table indicating the specific Activity and Finder index where the ResGrp is currently located by calling the *record_res_grp_progress*() function. It also creates a new record for this ResGrp in the [*SO_ResGrpCurrent*] state table by executing the SQL query shown in Equation (4).

INSERT INTO [SO_ResGrpCurrent] (4)

VALUES(<ResGrpID>, <SpecificActivityID>, 2, <FinderIndex> ,0,0,0)

where query parameters are shown in <> brackets, and the value 2 indicates that the ResGrp is currently located at a Finder (other possible values are 1 for Feeder and 3 for Sender).

The *FinderMatchID* parameter in Table 5 is an important variable that determines the major steps that the Finder will carry out to match Resources to Slots in building the ResGrp. The available options for this parameter are defined in the [*Def FinderMatch*] table.

If the value of the *FinderMatchID* parameter is 1 (termed a Type 1 Finder), the Finder carries out three major operations: the creation of a list of required Slots, the creation of a list of Candidate Resources, and the matching of the Candidate Resources to the required Slots to build the ResGrp. If the *FinderMatchID* parameter is 2 (Type 2 Finder), the Finder creates a list of Candidate Resources and uses the list as is to build the ResGrp without matching them to a separate list of required Slots. FinderMatchID values of 3, 4 and 5 are available only for Type 2 Activities and will be discussed in the next section on the Type 2 Activity Finder Process.

For Type 1 Finders, the Finder first executes code for generating the required Slot list, which has been wrapped in the function *build_slot_list()*. This function searches within specified Organizations and ResGrps to generate a list of Slots that represent the Resource requirements of the Activity. This list of Slots can be filtered and limited to a certain number of Slots by the user. The function returns a value indicating whether required Slots were found. If Slots are found, the Function stores them in the [*SO_ReqSlots*] intermediate table. Some post processing of the final Slot list is then carried out to store the Slot Attribute requirements for later use during the matching process and to remove certain Slots for whom a Resource will be automatically generated rather than being selected from the Establishment. If no Slots are found, the Finder has no further steps to carry out and the Activity proceeds to the next Finder in the list.

Once the Finder's required Slot list has been determined, the code attempts to fill the Slots with Establishment Resources. To do this, the code processes a Resource Candidate supply list which contains source Organizations and ResGrps that will supply Resource Candidates that can be used to fill the Required Slots. This list is divided into two categories, Primary and Augmentee, and multiple levels per category. The Primary supply list contains the primary sources that resources should be drawn from to fill the required Slots. The Augmentee supply list contains the secondary sources that resources can be drawn from if the Primary list fails to completely fill the required Slots. The levels within the Primary and Augmentee supply lists control the order in which the sources will be searched in attempting to fill the required Slots. In general, the search for Candidate resources follows the sequence shown in Equation (5).

$$P1, P2, \ldots, Pn, A1, A2, \ldots, Am \tag{5}$$

where *Pi* refers to the i^{th} level out of *n* total levels of the Primary list, and *Ai* refers to the i^{th} level out of *m* total levels of the Augmentee list.

For each category and level following the sequence in Equation (5), the Finder retrieves the Candidate Resources from the supply Organizations and ResGrps and then attempts to match the

Candidate Resources to the Required Slots. The code for retrieving the list of Candidate Resources is wrapped in the function *build_resource_list()*. The list of Candidates can be filtered and limited to a certain number. The function returns a value indicating whether Resources were found and how the code should proceed. If Resources are found the function stores them in the [*SO_SelResources*] intermediate table.

Once the Candidate Resource list has been generated for the current category and level being processed, the code looks at each as yet unfilled Slot in the required Slot list and attempts to match a Candidate Resource to it. The matching process is based on the Attribute requirements of the Slot and the Attributes of the Resources. If a match is found, the matched Slot and Resource are added to the Finder's ResGrp and removed from the Required Slot list and Candidate Resource list. Once every Slot in the Required Slot list has been tested for a match, if unfilled Slots remain, the code proceeds to the next level or category in the sequence in Equation (5), and repeats the process of searching for Candidates and matching them to unfilled Slots.

This process stops when all required Slots are filled or when the Resource Candidate supply list has been exhausted. In the latter case, the remaining unfilled Slots are added to the ResGrp and marked as unfilled. This completes the Finder's processing and the code then proceeds to the next Finder in the current Activity.

For Type 2 Finders, the ResGrp is created directly from the Resource Candidate list without matching the Resources to a separate list of required Slots. In this case, the user specifies a single Resource Candidate supply list from which the Finder builds the list of Resource Candidates by executing the *build_resource_list*() function. The total number of Resource Candidates is then limited to a certain value and the remaining Resources are used to build the ResGrp.

The following subsections provide detailed descriptions of the creation of the required Slot List, the creation of the Resource Candidate list, the creation of the ResGrp by matching, and the creation of the ResGrp from the Candidate list.

Required Slot List Creation

The *build_slot_list()* function is called by the Type 1 Finder process to generate a list of required Slots that the Finder will attempt to fill to meet the Resource requirements of an Activity. The function accepts the parameters *ResGrpID*, *ResReqID*, *FinderRecNum*, *TotalLimitTypeID*, *TotalLimitValue* and *SlotSortID* from the current Finder process. It returns an integer value that indicates whether Slots were found so that the Finder process knows how to proceed. A return value of 0 indicates that required Slots were found and the Finder should then attempt to fill them. The function stores the list of required Slots in the intermediate table [*SO_ReqSlots*]. Return values of 1, 2 and 3 are accepted by the Finder process to direct subsequent actions if no slots are found. Currently, if the *build_slot_list*() function does not find any slots, the only value it returns is 3 indicating that no further action is required by the Finder and the Activity should proceed to the next Finder in the list.

The *build_slot_list()* function begins by clearing the intermediate table [*SO_ReqSlots*]. The code then executes the stored query *SO_FinderResList* passing in the *ResReqID* as a parameter to retrieve the list of Organizations and ResGrps that contain the Required Slots. Typically, Theatre Organizations are specified in this list as Activities typically populate Theatre positions with

Establishment Resources. However, the flexibility is provided to identify any Organization or ResGrp that can provide a list of Slots.

If no Organizations or ResGrps are found by the query, the function records the message "Activity Finder Found no Orgs/ResGrps for Requirement List" to the [*Simulation Progress*] table and exits returning a value of 3 indicating that no Slots were found. If Organizations and/or ResGrps are found, the code loops through each Organization or ResGrp record.

For each Organization or ResGrp record, the code gathers the information shown in Table 6 that determines how it should process the associated Slots:

Table 6: Description of required Slot List parameters

Parameter	Description
OrgID	The Organization in which to search for Slots
ResGrpID	The ResGrp in which to search for Slots
LimitTypeID	The type of limit to be applied to the Slot list, for example, to limit the number of Slots selected to a certain value.
LimitValue	The specific value to be used by the limit type.
FinderResListRecNum	The ID of the Finder Resource Requirement List used to determine which filters to apply to the Slot list.

The code then clears the intermediate table [*SO_RawSlots*] which is used to store the un-processed list of Slots. If an OrgID was specified above, the code executes the query *SO_OrgSlots* which retrieves all Slots that belong to the specified OrgID and appends them to the [*SO_RawSlots*] table. This includes searching for Slots in all sub-organizations falling under the specified OrgID in the Organization tree structure. If a ResGrpID was specified above, the code executes the query *SO_ResGrpSlots* which retrieves the Slots contained in the specified ResGrp and appends them to the [*SO_RawSlots*] table. If both an OrgID and a ResGrpID were specified in the same record, the OrgID is used and the ResGrpID is ignored. If neither an OrgID nor a ResGrpID was specified above, the code proceeds to the next Organization or ResGrp record to search for Slots.

The next stage in building the Slot list is to process the [*SO_RawSlots*] table to determine those slots that will be added to the final Slot requirement list stored in the [*SO_ReqSlots*] table. First, the code loops through each Slot in the [*So_RawSlots*] table checking each Slot individually for certain requirements.

MARS allows the user to set up initial conditions by pre-filling ResGrps with Resources that are already known from current CF data or the CF planning process. However, such a ResGrp may not have all the Resources it requires at the time the simulation is run. The user may still intend

the simulation to create and fill additional Slots in this ResGrp. These additional Slots discovered in the preceding steps must be checked to ensure that they are not already present in the ResGrp as part of the initial conditions set up by the user. Any such duplicates in the [*SO_RawSlots*] table are removed, and the number of duplicates removed is stored in the *NumPrefills* variable.

In the next step, the code filters out Slots whose Attributes do not meet certain criteria defined by the user. These criteria are specified by the *FinderResListRecNum* variable which points to a set of Attribute Requirements in the [*Resource Requirement Filters*] table. Each Attribute Requirement is defined in the [*AttrReq Values*] table. An Attribute requirement consists of one or more rules that define permissible values for the Attribute. Each rule consists of an Attribute, a logical operator, and a value. The logical operator can be any one of the standard set of equality and inequality constraints (=, <>, <, <=, > and >=). The individual rules in an Attribute requirement are related by a logical-and or logical-or, for example "Resource Count >= 0 AND Resource Count <= 5." Slots that do not satisfy all the Attribute requirements specified by the *FinderResListRecNum* are removed from the [*SO_RawSlots*] table.

The Slot filter step is carried out in two steps. First, the Attribute requirements whose rules are related by a logical-or are evaluated using the stored query *SO_SlotFilterTESTOR*. This query ensures that at least one rule in each Attribute requirement is satisfied by the Slot's Attributes. If the Slot fails this test, it is removed from the [*SO_RawSlots*] table. If the Slot passes this test, the remaining Attribute requirements whose rules are related by a logical-and are evaluated using the stored query *SO_SlotFilterTESTAND*. This query ensures that all rules in each attribute requirement are satisfied by the Slot's Attributes. If the Slot fails this test, it is removed from the [*SO_RawSlots*] table.

Once all the Slots have been tested against the Attribute requirements, the code enforces a limit on the number of Slots selected based on the *LimitTypeID* and *LimitValue* variables. Any one of three limit types may be selected or none at all. If the *LimitTypeID* is set to zero, no limits are enforced. Otherwise, the code prepares the [SO_RawSlots] table to randomly select Slots to be removed to meet the limit. First a random integer, X where $0 \leq X < 100000$, is inserted into the *Order* field of the [*SO_RawSlots*] table. The [*SO_RawSlots*] table is then opened and ordered by the *Order* field. Depending on the type of limit to be applied, the number of Slots to be removed (if any) to enforce the limit is calculated. A *LimitTypeID* of 1 corresponds to a percentage limit. In this case the *LimitValue* indicates the percentage of Slots to be retained. The number of excess Slots to be removed for a percentage limit is calculated using Equation (6), below.

$$\text{Excess} = \text{Int}(\text{NumRecords} * (1 - \text{LimitValue})) - \text{NumPrefills} \quad (6)$$

where *NumRecords* is the number of Slots currently in [*SO_RawSlots*].

A *LimitTypeID* of 2 corresponds to a number limit. In this case the *LimitValue* indicates the maximum number of Slots that can selected from the current Organization or ResGrp. The number of excess Slots to be removed for a number limit is calculated using Equation (7), below.

$$\text{Excess} = \text{NumRecords} - \text{LimitValue} - \text{NumPrefills} \quad (7)$$

A *LimitTypeID* of 3 corresponds to a chance limit. In this case the *LimitValue* indicates the probability that each Slot in [*SO_RawSlots*] will be retained. The number of excess Slots to be removed is calculated by first executing the stored query *SO_SlotRemoveProb* which selects those Slots from [*SO_RawSlots*] whose random *Order* field value is less than (1 - LimitValue) * 100000. For example, if *LimitValue* is set to 0.85, approximately 15% of the Slots will be selected by this query. The excess to be removed is then the number of records selected by this query minus *NumPrefills*.

Once the excess has been calculated by any one of the three methods, if the number of excess Slots is positive, the code removes this number of Slots from [*SO_RawSlots*] proceeding in the order of the *Order* field.

Finally, the remaining set of Slots in [*SO_RawSlots*] is appended to the required Slots list in the [*SO_ReqSlots*] table. The code then repeats this process for the next Organization or ResGrp in the Finder's list, each time appending the unremoved Slots to the [*SO_ReqSlots*] table.

Once all the identified Organizations and ResGrps have been processed, the [*SO_ReqSlots*] table contains the total list of required Slots for the Finder. The Finder parameters listed in Table 5 contain a *TotalLimitTypeID* variable and a *TotalLimitValue* variable. The code uses these values to enforce an overall limit on the number of Slots in the [*SO_ReqSlots*] table. The implementation of the total limit is identical to the implementation of the local limits applied to the [*SO_RawSlots*] table while processing the individual Organizations and ResGrps.

The final step in the generation of the Required Slot list is to sort the [*SO_ReqSlots*] table. The sorting of the Slots determines the order in which the simulation will later attempt to fill them. The typical strategy is to sort the Slots so that the most difficult-to-fill Slots are filled first. This generally ensures that a larger proportion of Slots will be successfully filled by the Finder. The sorting is achieved by writing a value into the *Order* field of the [*SO_ReqSlots*] table for each Slot. The Finder will later attempt to fill the Slots in the order defined by the *Order* field.

The *SlotSortID* variable in Table 5 indicates the type of sorting to apply to the [*SO_ReqSlots*] table. A *SlotSortID* of 0 indicates that the Slots will not be explicitly ordered and will be filled in the order in which they currently appear in the [*SO_ReqSlots*] table which is out of the control of the user. A *SlotSortID* of 1 indicates that the Slots should be randomly sorted. In this case, a random integer, X where $0 \leq X < 100000$, is inserted into the *Order* field for each Slot.

A *SlotSortID* of 2 or 3 indicates that the sort order will be either ascending or descending, respectively, based on weights associated with each Slot's Attribute values. Slot Attributes are actually Attribute requirements that specify the Attributes that a Resource must have in order to fill that Slot. All Attribute requirements are listed in the [*Def AttrReq*] table which provides an *AttrReqWeight* field. This field stores a user-defined value that subjectively represents the expected difficulty of finding Resources that satisfy the Attribute requirement. For this weighted sort method, the sum of the weights for all the Slots Attribute requirements for each Slot is used to sort the Slots. The first step is to calculate the sum of the weights for each Slot by executing the stored query *SO_SlotAttrWeightSum*. This stores each SlotID with its sum of weights in the [*SO_WeightedSum*] intermediate table. Then the sum of weights for each Slot is copied to the Order field in the [*SO_ReqSlots*] table using the query *SO_SlotAttrWeightSumUpdate*.

A *SlotSortID* of 4 indicates a special case in which the Slots are sorted based on only the weight of each Slot's Rank Attribute requirement. In this case the code executes the query *SO_SlotSingleAttrWeightSum* passing in a parameter value of 1 which is the ID of the resource Rank Attribute. The query determines the weight of the Rank Attribute requirement for each Slot, storing it to the [*SO_WeightedSum*] intermediate table. As in the previous sort method, the stored query *SO_SlotAttrWeightSumUpdate* copies the weight value from the [*SO_WeightedSum*] table to the *Order* field of the [*SO_ReqSlots*] table for each Slot.

At this point, the creation of the Finder's required Slot list is complete and the *build_slot_list()* function returns to the main Finder VBA code block. Before the code proceeds to building the Resource Candidate list, some additional post-processing of the required Slot list is performed including resolving probabilistic Attribute requirements and creating auto-generated Resources.

Probabilistic Attribute requirements are determined randomly during simulation execution. For example, a Slot may have a 30% chance that it will be filled by a Reserve Force member and a 70% chance that it will be filled by a Regular Force member. Whether the Slot will require a Reserve or Regular Force member must be randomly determined at this point in the Finder process and saved so that the requirement will be used later during the matching process.

Auto-generated Resources are types of Resources can be treated as always being available. This assumption removes the need to include these Resources in the Establishment. Certain Attribute requirements can be set as auto-generating such that any Slot that has this Attribute requirement will have an auto-generated Resource automatically created to fill it.

To carry out these operations, the code first executes the SQL query shown in Equation (8) that adds a record to the [*ResGrp SearchPaths*] table that will contain any Slots that will be filled by auto-generated Resources.

INSERT INTO [ResGrp SearchPaths](ResGrpID, PriOrAugm, SeqLevel, Replication) VALUES(<ResGrpID>, 5, 1, <replication>) (8)

where query parameters are shown in <> brackets and the value 5 is the ID of the auto-generated search path.

The Finder code then clears the intermediate table [*SO_CloneSlotAttrReq*] that will store the Slot Attribute requirements after the probabilistic Attribute requirements have been determined. For each Slot in the [*SO_ReqSlots*] table, the code executes the stored query *SO_SlotResAttrReqs2* which retrieves the Slot's attribute requirements storing them in the [*SO_AttrReqList*] table. Each record in the [*SO_AttrReqList*] table is an Attribute requirement rule. A rule consists of an Attribute, an operator, and a value, for example *Component = Regular Force*. If the value in the *VariableID* field of the [*SO_AttrReqList*] table is 2, then the value used in the rule is to be determined by a probability distribution. The *AttrReqProbID* field indicates which Attribute requirement probability distribution to use. The available probability distributions are stored in the [*AttrReq Probabilities*] table and identified by the *AttrReqProbID*.

The Finder code then processes each rule in the [*SO_AttrReqL*ist] table. If it finds a rule whose value is to be determined probabilistically, it retrieves the probability distribution by executing the SQL query in Equation (9).

```
SELECT * FROM [AttrReq Probabilities]
WHERE AttrReqProbID = <AttrReqProbID>.
```
(9)

The code then generates a random number and determines which value in the probability distribution it corresponds to. This value is then written back into the *AttrValue* field of the [*SO_AttrReqList*] table. The code repeats this process for each Attribute requirement rule for the Slot.

The Slot's Attribute requirements in the [*SO_AttrReqList*] table are then checked to see if any of them indicate that the Slot should be filled by an Auto-generated Resource. The code then executes the stored query *SO_SlotAutoGen* which retrieves Attribute requirement records from the [*SO_AttrReqList*] table that are Auto-generating. If the query returns records, then the Slot will be filled by an Auto-generated Resource. Auto-generated Resources are created by appending a record to the [*Output_ResourceAutoGens*] using the SQL query in Equation (10).

```
INSERT INTO [Output_ResourceAutoGens](AutoGenID, AttrID, AttrValue)
VALUES(<AutoGenID>, <AttrID>, <AttrValue>)
```
(10)

where the <AttrID> and <AttrValue> parameters indicate the Attribute requirement that led to the Slot being filled by an Auto-generated Resource.

The Slot and its Auto-generated Resource are then added to the ResGrp by appending a record to the [*ResGrp SelectionResults*] table using the SQL query in Equation (11).

```
INSERT INTO [ResGrp SelectionResults]
VALUES(<RGSearchRecNumGen>, <ReqSlotID>, <AutoGenID>)
```
(11)

where the <*RGSearchRecNumGen*> parameter is the ID of the ResGrp's Auto-generated search path in the [*ResGrp SearchPaths*] table.

Slots with Auto-generated Resources are then considered filled and are removed from the required Slot list in [SO_ReqSlots].

Finally, if the Slot has not been filled by an Auto-generated Resource, its Attribute requirements are saved to the [*SO_CloneSlotAttrReq*] table for use later during the matching process by executing the stored query *SO_AppendSlotCloneAttrReqs*.

This process is repeated for all Slots in the [*SO_ReqSlots*] table. Once all Slots have been processed, the Finder proceeds to the creation of Resource Candidate lists and the matching of Candidate Resources to the required Slots.

Candidate Resource List Creation

For Type 1 Finders, the Finder process attempts to the fill the Slots in the required Slot list with Candidate Resources from the Establishment to create the ResGrp. It processes the Slots in the order defined by the *SlotSortID* parameter and the weights that were calculated and stored in the *Order* field of the [*SO_ReqSlots*] table. Typically, the Slots are processed in descending order (*SlotSortID* = 3) so that matching is attempted on the most difficult-to-fill Slots first.

For Type 2 Finders, the Finder process creates a Resource Candidates list and uses it as is to create the ResGrp. In this case, there is only one category and level specified (primary, level 1) for the creation of the Candidate list.

The code searches for Candidate Resources within a list of source Organizations and ResGrps. The source Organizations and ResGrps are processed in the sequence shown above in Equation (5). For a given category and level in the sequence, the code calls the function *build_resource_list*() to generate the list of Candidate Resources. This function takes the *ResGrpID*, the category (Primary or Augmentee), the level, the *ResCanListID*, the *FinderRecNum*, a *ResourceSortID*, and the *SpecTaskID* as parameters. It returns a value indicating how the Finder process should proceed. A return value of 1 indicates that no Candidate Resources were found and that the Finder should proceed to the next level in the sequence. A return value of 2 indicates that no Candidate Resources were found and that the Finder should proceed to the next category, i.e. switch from Primary to Augmentee. A return value of 3 indicates that no Candidate Resources were found and that there are no more categories or levels to search so the Finder should terminate the search for Resource Candidates. Any other return value indicates that Candidate Resources were found and that the Finder should proceed to attempt to match these Resources to any as yet unfilled Slots.

The first step in the *build_resource_list*() function is to retrieve the list of source Organizations and ResGrps by executing the stored query *SO_RotoOrg*. If the query finds no Organizations or ResGrps, then the function exits with a return value of 2. Otherwise, the code proceeds to process each item returned by the query.

First the code retrieves parameters that determine how the source Organization or ResGrp will be processed. These parameters are shown in Table 7.

Table 7: Description of Candidate Resource List parameters

Parameter	Description
ResCanRecNum	The unique ID of the Resource Candidate source item.
SelOrgID	The ID of the Organization that will supply Resource Candidates.
SelResGrp	The ID of the ResGrp that will supply Resource Candidates
SelLimitTypeID	The type of limit to apply to the Resource Candidates
SelLimitValue	The specific value of limit that will be used by the limit type.

The code then clears the [*SO_RawResources*] intermediate table so that it can be populated with the initial unprocessed list of Resources from the specified Organization or ResGrp. If *SelOrgID* is specified, the code executes the stored query *SO_OrgResources* which populates the [*SO_RawResources*] table with all the Resources that fall under the specified Organization. This includes Resources that belong to sub-organizations below the specified Organization. If *SelResGrp* is specified, the code executes the stored query SO_ResGrpResources which populates the [*SO_RawResources*] table with all the Resources that are part of the specified ResGrp.

The code then filters the Resources in the [*SO_RawResources*] table, removing those whose Attributes do not meet specified criteria. First, the [*SO_AttrReqList*] is cleared, then the code executes the stored query *SO_ResCandFilterReqs* which retrieves the Candidate Resource Attribute requirements based on the *ResCanRecNum* variable and stores them in the [*SO_AttrReqList*] table. Each Resource in the [*SO_RawResources*] table is then processed individually. First, the code checks if the Resource is already in the Finder ResGrp. This can occur, for example, if the Resource was pre-assigned to the ResGrp in the initial conditions, or if the Resource was assigned to the ResGrp during the matching process of a previous category or level. The query *SO_ResInResGrp* searches for the current Resource in the ResGrp. If the Resource is found, it is removed from the [*SO_RawResources*] table. The code then tests the Resource's Attributes against the Attribute requirements in the [SO_AttrReqList] table by executing the function *filter_resource*().

The *filter_resource*() function first verifies that the Resource has all the Attributes that will be tested by the Attribute requirements by executing the stored query *SO_ResourceAttrVerify*. The query returns the IDs of Attributes that the Resource is missing. If missing Attributes are found, the function exits, returning the ID of the first missing Attribute. If the Resource is not missing any Attributes, the code proceeds to test the Resource's Attributes against the Attribute requirements that contain rules related by a logical-OR by executing the stored query *SO_ResourceFilter6OR*. This query returns the IDs of the Resources Attributes that failed to meet the Attribute requirements. If failed Resource Attributes are found, the function exits, returning the ID of the first failed Attribute. If no failed Resource Attributes are found, the code tests the Resource's Attributes against the logical-AND-based Attribute requirement rules by executing the stored query *SO_ResourceFilter5AND*. This query returns the IDs of Resource Attributes that failed to meet the Attribute requirements. If failed Resource Attributes are found, the function exits, returning the ID of the first failed Attribute. If no failed Attributes are found, the code exits the function returning 0 to indicate that none of the Resource's Attributes failed to meet the Attribute requirements.

When the *filter_resource*() function returns, if the return value indicates the ID of an Attribute that failed to meet the filter Attribute requirements, then the Resource is removed from the [*SO_RawResources*] table. This process of verifying that the Resource is not already in the ResGrp and testing it against the filter Attribute requirements is repeated for each Resource in the [*SO_RawResources*] table.

The code then performs a test on all the Resources in [*SO_RawResources*] to determine if assigning them to the current Activity would conflict with Activities to which the Resources are already assigned. This test is called the Resource Utilization Level (RUL) test. It ensures that Resources are not assigned to too many activities at once. The RUL attribute is used to track Resource utilization. In MARS V2, Resources which are assigned to Activities are either utilized

at the 100% level (RUL = 1) or the 0% level (RUL = 0). For example, a training activity fully occupies the time of participating resources for the duration of the training and would therefore increment the RUL of its Resources to 1. On the other hand, a waiver activity which applies a waiver status to assigned Resources for the duration of the activity does not occupy the time of its resources and would therefore not alter their RUL level. At any given time, a Resource can be assigned to only one activity which has an RUL of 1 but it can be assigned to any number of activities with an RUL of 0.

The RUL test is performed by running the stored query *YC_RUL8_ResourcesOUT*. This query operates on the candidate Resources in the [*SO_RawResources*] table and produces a list of candidate resources for which assignment to the current ResGrp would result in their overutilization. The code then iterates through the result of the query, removing each identified resource from the [*SO_RawResources*] table.

The RUL test considers not only those Activities currently in progress but also any future Activities connected to them via Sender-Feeder connections. In other words, when a Resource is tested to determine its eligibility for a new ResGrp, the sequence of Activities in which the new ResGrp will participate can be determined. Additionally, some candidate Resources may already belong to ResGrps and the sequence of Activities that will occupy these existing ResGrps can also be determined. The RUL test determines if the timing of the new Activity sequence will conflict with any existing Activity sequences such that the RUL of the candidate resources will exceed 100%. The steps involved in identifying Resources that will encounter an RUL conflict are outlined below.

a) Determine the timing and RUL of the current and future Activities to which the candidate resources in [*SO_RawResources*] are already assigned.

b) Determine the timing and RUL of the Activity currently acquiring resources and its future Activities.

c) Determine those Activities from (a) which would cause RUL conflicts with those in (b).

d) Determine which resources in [*SO_RawResources*] are already assigned to the Activities found in (c). These are the Resources that are ineligible to be considered for assignment to the current ResGrp due to an RUL conflict.

The last step in preparing the raw Resources list is to limit the number of Resources selected from the current Organization or ResGrp that will be used as Candidates. The implementation of the limit on the [*SO_RawResources*] table is identical to the limit used on the [*SO_RawSlots*] table during the preparation of the required Slot list. In this case the *SelLimitTypeID* parameter determines the type of limit to be applied, for example a percentage limit, and the *SelLimitValue* parameter determines the value of the limit, for example 0.75 which would retain 75% of the Resources in the [*SO_RawResources*] table. The remaining Resources are appended to the [*SO_SelResources*] table by executing the stored query *SO_SelResourcesAppend*. This table stores the Resources that will be used as Candidates for the matching process.

When applying the limit on the number of Resources that will be selected from the [*SO_RawResources*] table, a fourth option for the *SelLimitTypeID* parameter is available in

addition to the three already discussed in the Section on preparing the required Slot list. This fourth option is termed "Select" and allows an arbitrary number of Resources to be randomly sampled from the [*SO_RawResources*] table and added to the final [*SO_SelResources*] table. In this process, the same Resource in the [*SO_RawResources*] table is allowed to be randomly selected more than once and added to the [*SO_SelResources*] table. This type of Resource limit is used exclusively in conjunction with Type 3 and 4 Finders that use the Candidate list as a template to create new Resources dynamically. In these cases, because each Resource record in the [*SO_SelResources*] table is used to clone a new resource with a unique ID, the possible appearance of duplicates in this table is not a problem. The Section below on the Activity Type 2 Finder Process describes how Type 3 and 4 Finders create a new Resource for each row in the [*SO_SelResources*] table. The combination of *SelLimitTypeID* 4 and Type 3 or 4 Finders allows the user to identify a source population using the Resource Candidate list and to create an arbitrary number of new Resources randomly sampled from the Candidate list.

This process of generating Resource Candidates is repeated for each Organization or ResGrp specified for the current category and level. The last step the code performs is to sort the final Candidate Resources list in the order that they should be processed when attempting to match them to the required Slots. The sort operation applied to the Candidate Resources in the [*SO_SelResources*] table is identical to the sort operation applied to the required Slots in the [*SO_ReqSlots*] table. In this case the *ResourceSortID* parameter which was passed into the *build_resource_list*() function determines the type of sorting applied to the Resources.

ResGrp Creation by Matching Candidate Resources to Required Slots

For Type 1 Finders, if the *build_resource_list*() function successfully generates a new list of Resource Candidates in the [*SO_SelResources*] table, the code attempts to match these Resources to the as yet unfilled Slots in the [*SO_ReqSlots*] table. First, the code adds a new record to the [*ResGrp SearchPaths*] table for the current ResGrp, category and level by executing the SQL query in Equation (12).

INSERT INTO [ResGrp SearchPaths](ResGrpID, PriOrAugm, SeqLevel, Replication) VALUES(<ResGrpID>, <PriAug>, <Level>, <replication>) (12)

The primary key value generated for this record, *RGSearchRecNum*, will be used to identify Resources that are matched from the current category and level.

For each Slot in the [*SO_ReqSlots*] table, proceeding in the order defined above, the code tests every Candidate Resource until a match is found. The Candidate Resources are tested in the order defined by the *ResourceSortID* and the *Order* field of the [*SO_SelResources*] table. Typically, the Resources are processed in ascending order, where the *Order* field contains the sum of the weights associated with their Attribute values (*ResourceSortID* = 2).

For a given Slot, the code first clears the [*SO_AttrReqList*] table and then populates it with the Slot's Attribute requirements from the [*SO_CloneSlotAttrReq*] by executing the stored query *SO_AppendSlotAttrReqs*. Then, for each Candidate Resource in [*SO_SelResources*], the code tests if the Resource's Attributes match the Slot's Attribute requirements by calling the function *filter_resource*(). This is the same function that was used to filter Resources during the creation

of the Resource Candidate list. If the Resource meets the Slot's Attribute requirements, the function returns 0. If the Resource fails to meet one or more of the Slot's Attribute requirements, the function returns the ID of the first Attribute that failed to meet the Slot's requirements.

The code then writes a record of the match attempt, whether it succeeded or failed, to the [*ResGrp SearchResults*] table by executing the SQL query shown in Equation (13).

```
INSERT INTO [ResGrp SearchResults] ( RGSearchRecNum, SlotID, SlotWeight,
ResID, ResWeight, FailedAttrID, [Replication] )
VALUES(<RGSearchRecNum>, <ReqSlotID>, <SlotWeight>, <CandResID>,
<ResWeight>, <TestAttrID>, <replication>)
```
(13)

If the match attempt succeeded, the Resource and the Slot are added to the ResGrp by executing the SQL query shown in Equation (14). The Slot is then deleted from the [*SO_ReqSlots*] table, and the Resource is deleted from the [*SO_SelResources*] table.

```
INSERT INTO [ResGrp SelectionResults]
VALUES(<RGSearchRecNum>, <ReqSlotID>, <CandResID>)
```
(14)

If the match attempt failed, the code proceeds to the next Candidate Resource and attempts the match again. If the match attempt succeeded, the code repeats the matching procedure on the next Slot, starting from the beginning of the Candidate Resource list. This process continues until either all the Slots are filled, or all attempts have been made to match the current Resource Candidates to the remaining unfilled Slots. In the former case, the matching process has completed. In the later case, the Finder proceeds to the next level or category in the Resource search sequence and builds a new list of Resource Candidates and attempts to match them to the remaining Slots. The Finder continues to make match attempts using each category and level in the Resource search sequence until either all Slots are filled or no more levels remain in the Augmentee category, at which point the matching process is complete and the Finder moves on to finalize the processing of the ResGrp.

The last step in preparing the ResGrp is to append any remaining unfilled Slots. The code first adds a record to the [*ResGrp SearchPaths*] table to identify the unfilled Slots by executing the SQL query in Equation (15).

```
INSERT INTO [ResGrp SearchPaths](ResGrpID,PriOrAugm,SeqLevel,Replication)
VALUES(<ResGrpID>, 4, 1, <replication>)
```
(15)

where the value 4 is the ID of the unfilled Slot path.

The primary key value for this record is stored in the variable *RGSearchRecNum* and is used to identify the unfilled Slots in the [*ResGrp SelectionResults*] table. The code then appends the unfilled Slots to the [*ResGrp SelectionResults*] table by executing the SQL query in Equation (16).

```
INSERT INTO [ResGrp SelectionResults] ( ResGrpSearchRecNum, SlotID, ResID)
SELECT <RGSearchRecNum> AS Expr1, SO_ReqSlots.SlotID, 0 AS Expr2
```
(16)

FROM SO_ReqSlots

At this point the Finder has completed the creation of the new ResGrp. The Finder then applies the Resource Attribute updating instructions which reflect the nature of the current Activity and the role the selected Resources will play in it. This is accomplished using the same three-step operation described for the Feeder process in which the three functions *setup_attr_update*(), *setup_resource_update*() and *attribute_update*() are called in sequence.

The final step for the Finder is to record the number of Resources found compared to the number of required Slots. The code counts the number of Resources and the number of Slots in the ResGrp and then updates the totals in the [*SO_PartTypeReqs*] table by executing the SQL query in Equation (17).

UPDATE SO_PartTypeReqs
SET NumSupplied = NumSupplied + <ResCount>,
NumTotal = NumTotal + <SlotCount>
WHERE PartType = <PartType> (17)

where the <PartType> parameter is specified by the Finder and indicates the type of Resource that the ResGrp is supplying to the Activity.

These Part Type counts are used later in the Test Firing Rules sub-model to determine if sufficient Resources were found to run the Activity. At this point the Finder process is complete and the code proceeds to the next Finder of the current Activity and repeats the Finder process. Once all Activity Finders have been processed, the Activity Entity proceeds to the Test Firing Rules sub-model.

ResGrp Creation from Candidate List

For Type 2 Finders, the Finder creates the ResGrp directly from the primary level 1 Resource Candidates list. Once the Candidate list has been created and stored in the [*SO_SelResources*] table, the *TotalLimitTypeID* and *TotalLimitValue* parameters from Table 5 are used to enforce a limit on the number of Resources in the [*SO_SelResources*] table. This limit is implemented in the same manner that was used to limit the final required Slot list above.

The Resources in the [*SO_SelResources*] table are then used directly to build the ResGrp by executing the stored query *SO_BuildResGrpFromRoto*. This query transfers the Candidate Resources and the Slots they currently occupy in the Establishment to the ResGrp. Because the Candidate list was used to create the ResGrp without matching to an independent list of required Slots, Type 2 Finder ResGrps never contain any unfilled Slots. The Finder then completes its processing by performing attribute updates and tallying the Part Type counts in the same manner as for Type 1 Finders. The Activity then proceeds to the next Finder in the list.

7.8 Process Type 2 Activity Finder

In the Process Type 2 Activity Finders sub-model, shown in Figure 19, Type 2 Activity Entities build ResGrps with Finders and carry out Attribute updating instructions on the selected

Resources. Type 2 Activities have zero-duration and do not receive or send ResGrps to other Activities through Feeders and Senders. The sub-model is implemented entirely in a single VBA block.

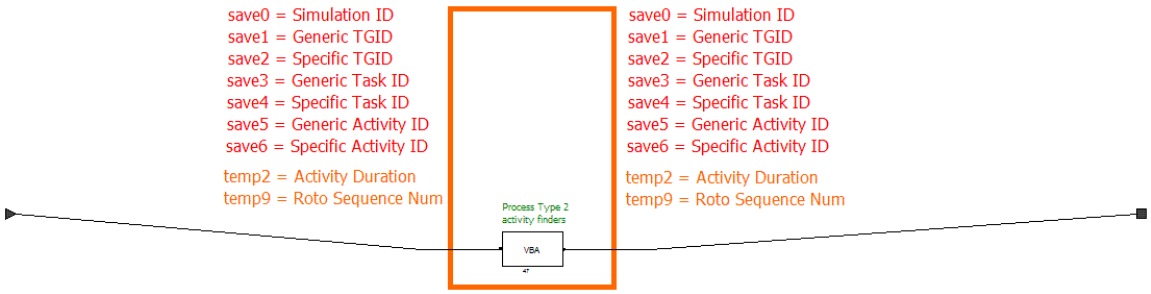

Figure 19: Process Type 2 Activity Finder sub-model.

The Type 2 Activity Finder process begins by following the same steps as a Type 1 Activity Finder. The Activity first accesses the list of its Finders by executing the stored query *SO_ActivityFinders*. The code then processes each Finder in the list in order of the Finder index.

For each Finder, the code first retrieves the Finder parameters listed previously in Table 5 that determine how the Finder will proceed. The code then determines the Finder's *ResGrpID* and *ResCanListID* by executing the stored query *SO_FinderResGrpID*.

The *FinderMatchID* parameter determines the major actions that the Finder will carry out. A *FinderMatchID* of 1, in which Candidate Resources are matched to a list of required Slots, is not available for Type 2 Activity Finders. As with Type 1 Activity Finders, when *FinderMatchID* is 2, the Candidate Resources are used directly to create the ResGrp.

*FinderMatchID*s 3 and 4 are used to create new resources in simulation scenarios that implement Establishment dynamics. *FinderMatchID* 3 generates a Candidate Resources list and then creates a new Resource for each Candidate, copying its Attributes. The newly created Resources are then used to build the ResGrp. *FinderMatchID* 4 performs the same steps as *FinderMatchID* 3, but performs the additional steps of removing the selected Candidate Resources from their current Slots and moving them into a special Transfer Slot. Each newly created Resource is then moved into the Slot vacated by the Candidate from which it was copied.

FinderMatchID 3 allows the user to create a special organization that is representative of the recruit population, or more generally, any Resource intake population. The finder then randomly samples within this representative population to dynamically create new Resources to enter the simulation.

The purpose of *FinderMatchID* 4 is to implement a simplified form of personnel turn over where a proportion of the current population transfers out of their current positions and is replaced by newly recruited or promoted Resources. It is the responsibility of the user to update the Attributes of the newly created Resources to reflect those of a newly recruited or promoted Resource, otherwise the new Resources will have exactly the same Attribute values as the Candidate Resource from which it was copied.

Regardless of the *FinderMatchID*, The Type 2 Activity Finder creates a list of Candidate Resources by calling the function *build_resource_list*(). The function processes the Primary level 1 list of source Organizations and ResGrps. As with Type 1 Activity Finders, the code then adds a record to the [*ResGrp SearchPaths*] table. The primary key value of this record, *RGSearchRecNum*, is used to identify the Resources that will be attached to the *ResGrp*.

If the *FinderMatchID* is 3 or 4, the code then uses the Candidate Resources in [*SO_SelResources*] to create new Resources. If the *FinderMatchID* is 4, the code clears the intermediate table [*SO_SelResourcesSave*] which will be used to store the Candidate Resources so their Attributes can later be updated to reflect moving them to the Transfer Slot. For each record in the [*SO_SelResources*] table, the code creates a new Resource by calling the function *create_resource*(). This function adds a new record to the [*Resources*] table giving it a new Resource ID number by executing the SQL query shown in Equation (18).

```
INSERT INTO Resources( ResID, ResTag, ResName, Rep )
VALUES (<NewResID>, <ResTag>, <ResName>, <Replication>)
```
(18)

The next step for *FinderMatchID* 3 and 4 Finders is to copy the Attributes from the Candidate Resource to the New Resource in the [*OUTPUT_ResourceAttributes*] record table by executing the stored query *SO_CopyAttrToNewRes*. The Attributes are also copied to the [*SO_ResAttrCurrent*] state table by calling the stored query *SO_CopyResAttrToCurrent*.

If the *FinderMatchID* is 4, the Candidate Resource is saved to the [*SO_SelResourceSave*] table by executing the SQL query in Equation (19).

```
INSERT INTO SO_SelResourcesSave(ResID)
VALUES(<SelResID>)
```
(19)

The code then replaces the ID of the Candidate Resource in the [*SO_SelResources*] table with the ID of the newly created Resource. This completes the creation of the new Resource for *FinderMatchID* 3 and 4 Finders. This process is repeated for each record in the [*SO_SelResources*] table.

For *FinderMatchID* 4 Finders, once all the new Resources have been created, the original Candidate Resources must vacate their Establishment Slots and move to the Transfer Slot. This is accomplished using Attribute update instructions. First the code clears the [*SO_AttrUpdate*] table. It then adds an attribute update instruction to the [*SO_AttrUpdate*] table that sets the Resource Slot Attribute to the ID of the Transfer Slot by calling the stored query *SO_AttrUpdateTransfer*. The code clears the [*SO_ResourceUpdate*] table and then appends the original Candidate Resource IDs from the [*SO_SelResourceSave*] table by calling the stored query *SO_ResourceUpdateTransfer*. Lastly the code calls the *attribute_update*() function which applies the Attribute update instruction to the original Candidates, moving them to the Transfer Slot.

Next, the Finder Attribute update instructions for the Selected Resources in the [*SO_SelResource*] table are applied. For *FinderMatchID* 2 Finders, these are the Resource Candidates. For *FinderMatchID* 3 or 4 Finders, these are the newly created Resources. This is accomplished using the three-step procedure described previously consisting of calling the *setup_attr_update*()

function to retrieve the Attribute update instructions, the *setup_resource_update*() function to retrieve the Resources to be updated, and the *attribute_update*() function apply the Attribute updates to the selected Resources.

Finally, the ResGrp is created using the Resources in the [SO_SelResources] table by calling the stored query *SO_BuildResGrpFromRoto*. This completes the Type 2 Activity Finder process, and the Activity then proceeds to its next Finder.

7.9 Activity Part Test

In the Part Test sub-model, shown in Figure 20, the Type 1 Activity Entity determines if it has acquired sufficient Resources through its Feeders and Finders to execute and employ the Resources for its duration.

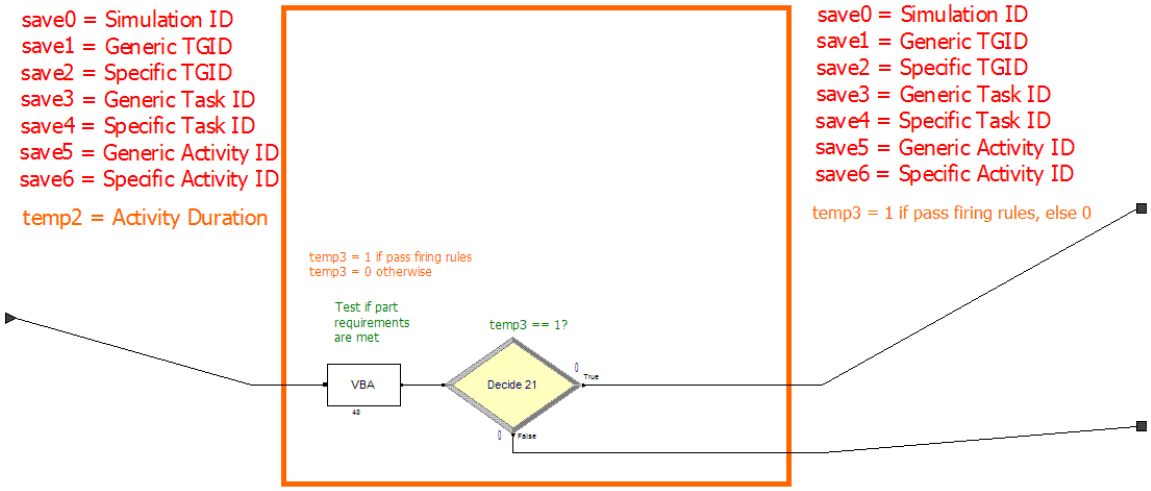

Figure 20: Part Test sub-model.

The Part Type test is implemented in a VBA block. The VBA code first opens the [*SO_PartTypeReqs*] table which the Activity's Feeders and Finders used to accumulate the number of Resources found and the number of Resources required for each Part Type. Each row in the table contains the data for a single Part Type. The code iterates through each row in the table verifying if the ratio of the number of Resources found to the number of required Resources is greater than or equal to the required percentage for that Part Type. If any Part Type fails to meet this requirement, the Activity entity fails the Part Type test. If all Part Types meet this requirement, the Activity passes the Part Type Test. The result of the Part Type test is saved to the *temp3* attribute. The attribute is set to 1 if the Activity passes the Part Type test or 0 if the Activity failed to meet one or more of the Part Type requirements.

After leaving the VBA block, the Activity entity passes through a decide block which directs the Activity depending on the value of its *temp3* attribute. If the *temp3* attribute is set to 1, the Activity proceeds to the Run sub-model which runs the activity, occupying its selected Resources for the duration of the Activity. If the *temp3* attribute is set to 0, the Activity skips the Run sub-model and is immediately sent to the Route ResGrps to Senders sub-model.

7.10 Run Activity

In the Run Activity sub-model, shown in Figure 21, the Activity entity is delayed by the duration specified in the Activity's *temp2* attribute. This holds the Activity's Resources in their current state until the delay period has passed, at which point the Activity entity proceeds to the Route ResGrps to Senders sub-model.

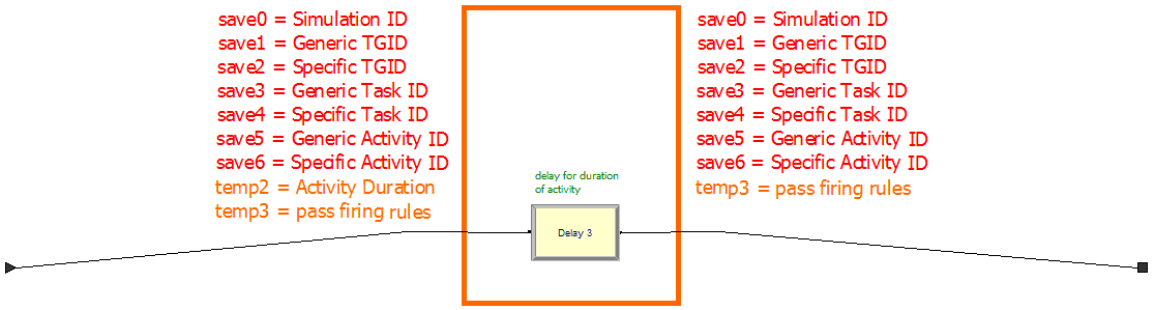

Figure 21: Run Activity sub-model.

7.11 Route ResGrps to Senders

In the Route ResGrps to Senders sub-model, shown in Figure 22, the Activity entity routes its ResGrps internally from its Feeder and Finder nodes to its Sender nodes and executes the Attribute updating instructions associated with the Senders. The ResGrps are routed differently depending on whether the Activity ran successfully or failed to run because it did not meet its Part Type requirements. This allows the ResGrps to receive different Attribute updating instructions and to be sent to different subsequent Activities depending on whether the Activity found sufficient Resources to run.

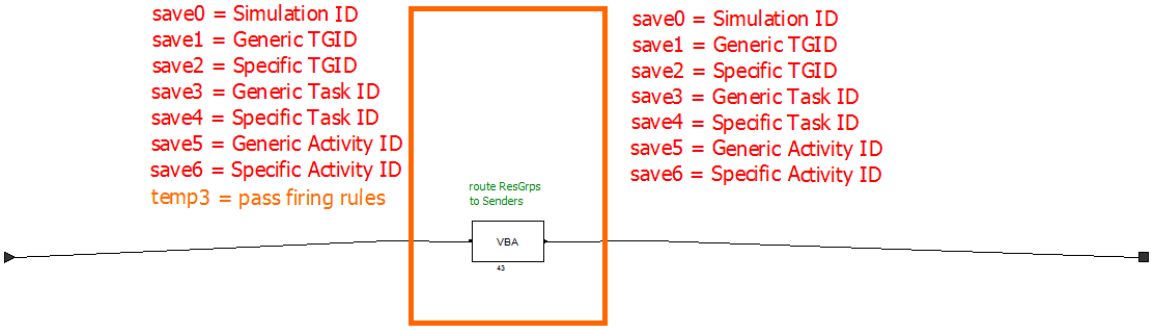

Figure 22: Route ResGrps to Senders sub-model.

The Route ResGrps to Senders sub-model is implemented in a single VBA block. The code begins by executing the stored make-table query *SO_RouteResGrptoSender*, passing in the specific Activity ID as a parameter, to create the [*TEMP_RouteResGrptoSender*] intermediate table. The query accesses the [*Activity Connections*] table to retrieve the Senders to which each ResGrp should be routed. It retrieves two Senders for each ResGrp, one to be used if the Activity ran successfully and one to be used if the Activity failed to run due to insufficient Resources.

The code then iterates through each row in the created [*TEMP_RouteResGrptoSender*] table. For each row, it retrieves the appropriate Sender index depending on the *temp3* attribute which indicates whether the Activity passed or failed the Part Type test. The code then moves the ResGrp to the specified Sender by executing the *set_res_grp_current*() function and records the move by executing the *record_res_grp_progress*() function. Finally, the code applies the Sender's Attribute updating instructions to the Resources in the ResGrp by executing the three Attribute update functions described previously: *setup_attr_update*(), *setup_resource_update*() and *attribute_update*(). The code then repeats these steps on the next ResGrp in the [*TEMP_RouteResGrptoSender*] table. When all Activity's ResGrps have been routed to Senders and updated, the Activity entity proceeds to the Route ResGrps to Next Activity sub-model.

7.12 Route ResGrps to Next Activity

In the Route ResGrps to Next Activity sub-model, shown in Figure 23, the Activity sends the ResGrps from its Senders on to subsequent Activities to which it is connected with a Sender-to-Feeder connection. The Activity then sends a signal to each receiving Activity indicating that a ResGrp has been sent. The sub-model is also responsible for sending signals to waiting Activities that depend on the current Activity. These signals will be received by Activities in the waiting queue in the Activity Queue sub-model. An Activity in the wait queue that receives one of these signals is triggered to check if it has now satisfied all the criteria in order to be released from the wait queue.

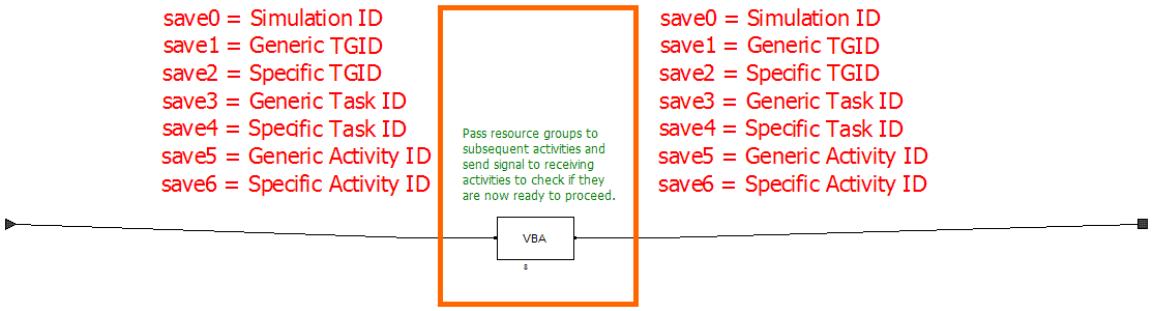

Figure 23: Route ResGrps to Next Activity sub-model.

The Route ResGrps to Next Activity sub-model is implemented in a single VBA block. The VBA code begins by executing the stored query *SO_ResGrpPass* passing the current replication, specific Task ID and specific Activity ID as parameters. This query accesses the [*Task Activity Links*] table to retrieve the specific Activity ID and Feeder index to which each ResGrp will be routed. The code then iterates through each row in the query result recordset, moving each ResGrp to the specified Activity Feeder by calling the *set_res_grp_current*() function. The receiving Activity is then sent a signal to tell it that a ResGrp has arrived at one of its Feeders. The code then repeats this process for each ResGrp in the *SO_ResGrpPass* query recordset.

Note that, the arrival of the ResGrps at the follow-on Activity Feeders is not recorded using the *record_res_grp_progress*() function at this point in Sender processing. This is because the time at which a ResGrp is acquired by an Activity Feeder is technically the time at which that Activity begins processing. The receiving Activity may still have to wait for other ResGrps to be sent, for other Activities on which it depends to finish or for its SNET time to arrive before it can begin

processing. Therefore, the arrival of the ResGrp at the follow-on Activity Feeder is not recorded until the Activity is released from the wait queue and begins processing its Feeders.

The last step in the Route ResGrps to Next Activity sub-model is to signal waiting Activities that depend on the current Activity. The code retrieves a list of dependent activities by executing the stored query *SO_SignalDependencies* passing the current specific Activity ID as a parameter. It then iterates through each row in the query result recordset, sending a signal to each dependent Activity.

This completes the processing of the Activity Senders, and the Activity entity proceeds to the Finish Activity sub-model.

7.13 Finish Activity

In the Finish Activity sub-model, shown in Figure 24, the Activity entity enters a VBA block and writes a record that it has finished processing and proceeds to the Wait for Activities to Finish sub-model.

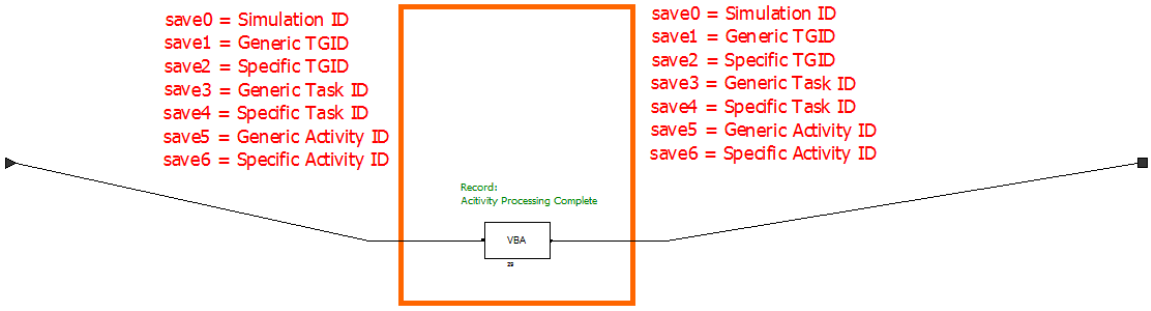

Figure 24: Finish Activities sub-model.

7.14 Wait for Activities to Finish

In the Wait for Activities to Finish sub-model, shown in Figure 25, the Activity Entity informs its parent Task Entity that it has finished processing and then exits the simulation.

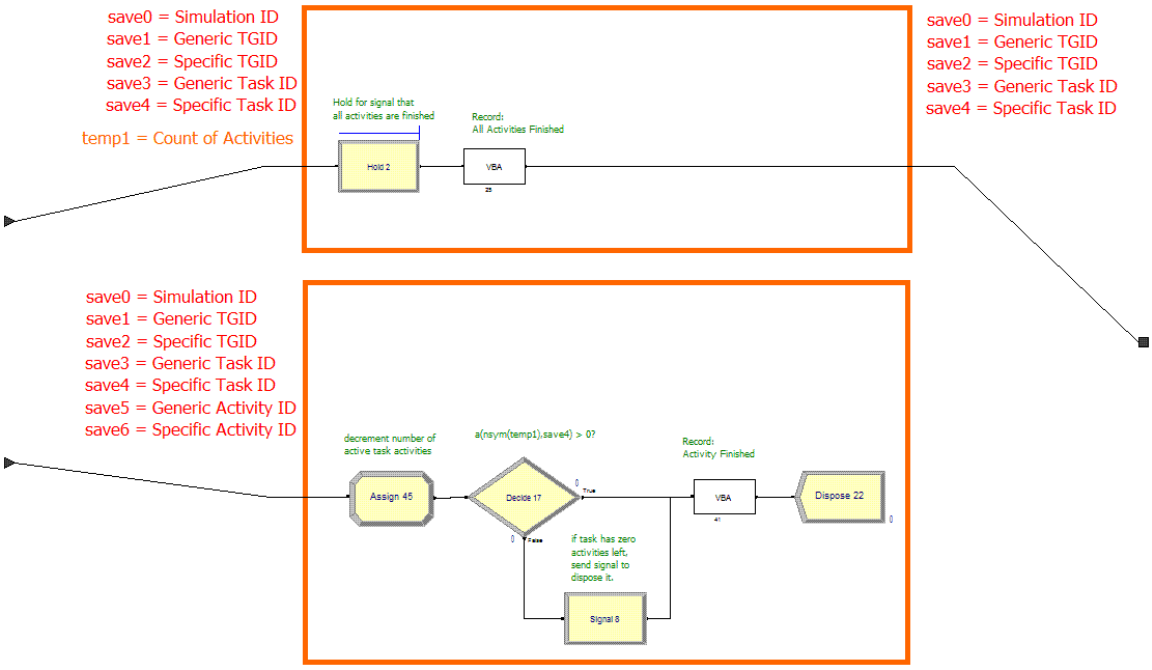

Figure 25: Wait for Activities to Finish sub-model.

The Activity entity enters the sub-model via the lower node on the left side of Figure 25. The Activity's parent Task entity entered the sub-model via the upper left node at the start of the simulation and has been waiting in the Hold block for its Activities to finish. The Task entity's *temp1* attribute initially contained the total number of Activities created as part of the Task. As the simulation runs, the *temp1* attribute tracks the number of as yet unfinished Activities.

When the Activity entity arrives, it enters an assign block which uses a special Arena function, shown in Equation (20), to access the *temp1* attribute of its parent Task entity and decrement it by one.

$$a(nsym(temp1),save4) = a(nsym(temp1),save4) - 1 \quad (20)$$

The function *a*() used in Equation (20) provides access to the attributes of any Arena entity. It takes two parameters; the first is the ID of the attribute (which is retrieved using the Arena function *nsym*() and passing the name of the attribute as a parameter), and the second is the ID of the entity. In this case, the ID of the parent Task entity is stored in the Activity entity's *save4* attribute.

Once the Task entity's *temp1* attribute has been decremented, the Activity entity enters a decide block that checks if Task's *temp1* attribute has reached zero. If this is the case, the Activity signals the Task entity, indicating that all the Task's Activities have now finished. The Activity entity then passes through a VBA block that records that the Activity has completed processing and then leaves the simulation at the dispose block.

If the current Activity entity was the last of the Task's Activities to finish, the Task entity will have been signalled, releasing it from the hold queue. The completed Task entity then passes through a VBA block which writes a record that all the Task's Activities have finished. The Task entity then proceeds to the Wait for Tasks to Finish sub-model.

7.15 Wait for Tasks to Finish

In the Wait for Tasks to Finish sub-model, shown in Figure 26, the Task entity informs its parent Task Generator entity that it has finished processing and then exits the simulation.

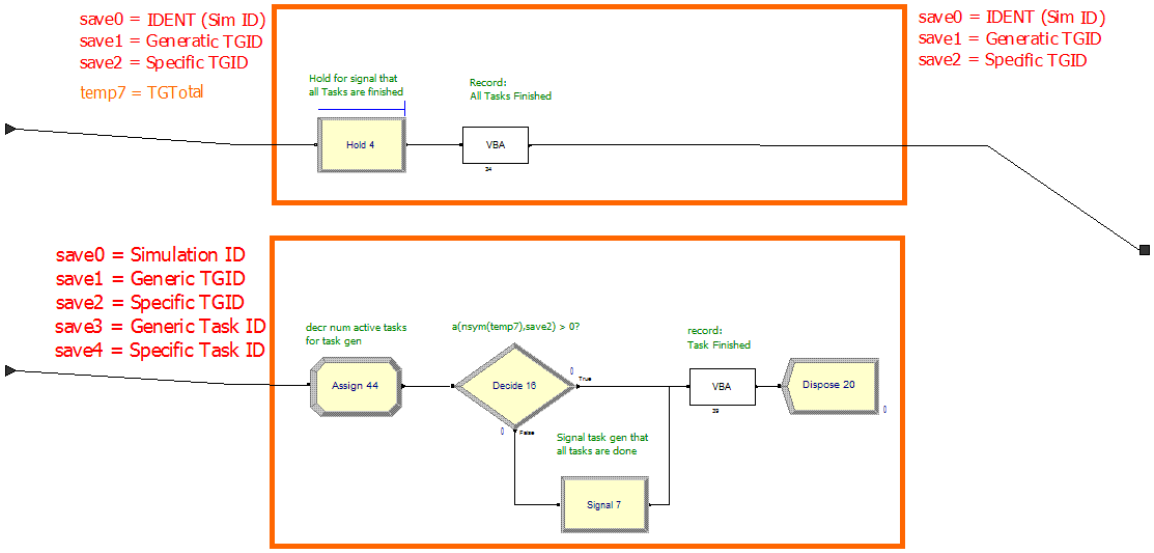

Figure 26: Wait for Tasks to Finish sub-model.

The structure of this sub-model is identical to the Wait for Activities to Finish sub-model except that the finishing Task entity communicates with its parent Task Generator entity, and the Task Generator entity uses the *temp7* attribute to track the number of as yet unfinished Tasks.

If the current Task entity was the last of the Task Generator's Tasks to finish, the Task Generator entity will have been signalled, releasing it from the hold queue to proceed to the Wait for Tasks Generators to Finish sub-model.

7.16 Wait for Task Generators to Finish

In the Wait for Task Generators to Finish sub-model, shown in Figure 27, the Task Generator entity informs the Simulation entity that it has finished processing and then exits the simulation.

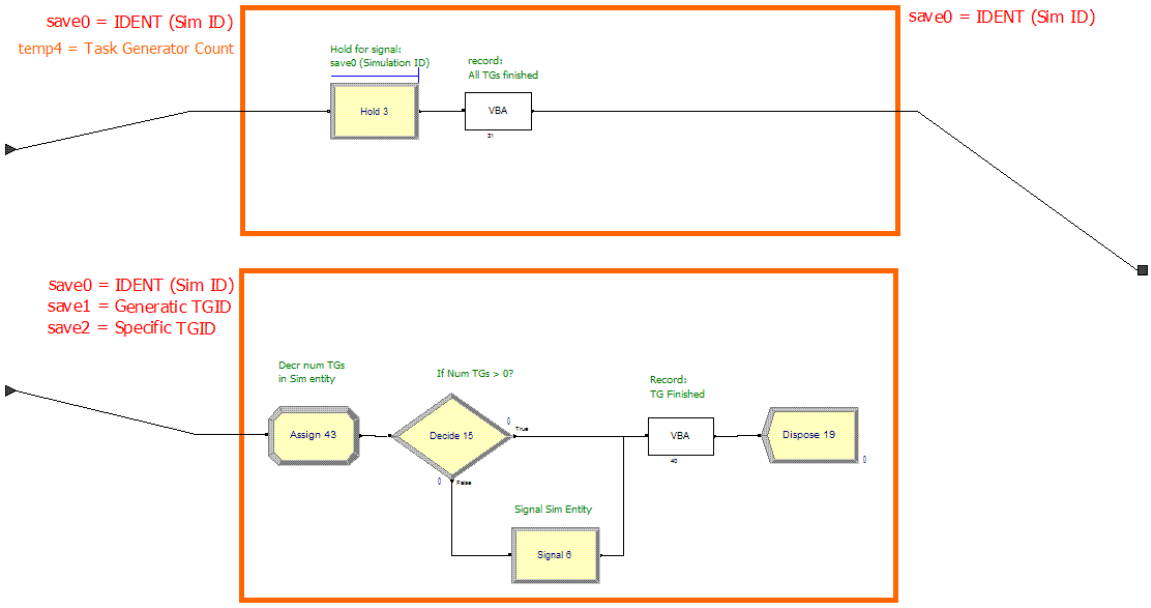

Figure 27: Wait for Tasks Generators to Finish sub-model.

The structure of this sub-model is identical to the Wait for Activities sub-model except that the finishing Task Generator entity communicates with its parent Simulation entity, and the Simulation entity uses the *temp4* attribute to track the number of as yet unfinished Task Generators.

If the current Task Generator entity was the last of the Simulation's Task Generators to finish, the Simulation entity will have been signalled, releasing it from the hold queue to proceed to the End Simulation sub-model.

7.17 End Simulation

In the End Simulation sub-model, shown in Figure 28, the Simulation entity enters a VBA block which records that the simulation is complete. It then exits at the dispose block ending the simulation.

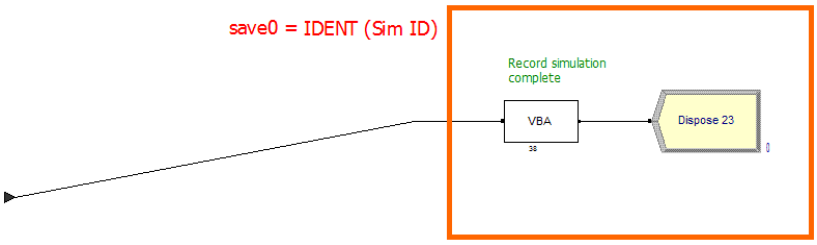

Figure 28: End Simulation sub-model.

8 Conclusion

This paper provided a detailed reference on the implementation of the MARS V2 managed readiness model. The managed readiness model is implemented in Arena using discrete event logic with embedded VBA code blocks. The VBA code provides an interface between the Arena model and the Simulation Runtime Database. A significant portion of the complex data operations that are part of the MARS model are implemented as SQL commands acting on the database and invoked from the VBA code.

Following the sequence of events in the simulation lifecycle, broken down into various sub-models, this paper documented how the Arena discrete event logic and the VBA code work to execute a simulation in MARS. The description of the implementation is meant to be thorough and detailed. It is targeted at analysts responsible for developing and maintaining MARS capabilities. It is possible to follow the sequence of Arena logic and VBA code from simulation start to finish in parallel with its description in this document. Users of MARS who build and run simulation scenarios may also refer to this document to verify exactly how some aspect of the model functions. Ultimately, it provides a complete reference on how MARS works at the lowest level which will assist future MARS developers and users in ensuring that the model continues to meet and adapt to client needs.

References

[1] Ormrod, M., Young, C., and Pall R. (2007). Modelling Force Generation with the Managed Readiness Simulator (MARS): Modelling Concept and Requirements for MARS v1.0. DRDC CORA Technical Memorandum TM 2007-65.

[2] Kelton, D.W., Sadowski, R. P., Sturrock, D. R. (2004) Simulation with Arena. 3rd ed. Boston: McGraw-Hill Higher Education, 2004.

[3] Pall, R., Young, C., and Ormrod, M. (2007). Modelling Force Generation with the Managed Readiness Simulator (MARS): Implementation of MARS v1.0 in a Discrete Event Simulation Environment. DRDC CORA Technical Memorandum TM 2007-52.

[4] Young, C., Pall, R., and Ormrod, M. (2007). A Framework & Prototype for Modelling Army Force Generation. DRDC CORA Technical Memorandum TM 2007-54.

[5] Ormrod, M., Young, C. (2007). Preliminary Analysis of Task Force Afghanistan Sustainability Using MARS. DRDC CORA Technical Memorandum TM 2007-40

[6] Okazawa, S., Ormrod, M., Young, C. (2009). Managed Readiness Simulator (MARS) V2: Assessment of a Simulation Runtime Database Approach. DRDC CORA Technical Memorandum TM 2009-043.

[7] Okazawa, S., Ormrod, M., Young, C. (2009). Managed Readiness Simulator (MARS) V2: Design of the Managed Readiness Model. DRDC CORA Technical Memorandum TM 2009-057.

Annex A MARS Database Tables

Table role legend:

D = Scenario definition

E = External data transfer

I = Intermediate results

O = Output and post processing

R = Simulation record

S = Simulation State

X = Unused/obsolete

Table name	**Role**
Activity Connections	D
Activity Firing Rules	D
Attribute Inheritance	D
AttrReq Probabilities	D
AttrReq Values	D
DBHistory	D
Def ActivityType	D
Def Attribute Values	D
Def Attributes	D
Def AttrReq	D
Def AttrReqCompare	D
Def AttrReqProb	D
Def AttrReqProcess	D
Def AttrValuType	D
Def Augmenteeable	D
Def Distributions	D
Def FeederFinderSender	D
Def FinderMatch	D
Def LimitType	D
Def MsgType	D
Def Object	D
Def PrimAug	D
Def Probabilities	D
Def ResReq	D
Def ResSource	D

Def Runtime UseQuals Options	D
Def SetAttrProb	D
Def SetAttrType	D
Def SimProgressMsg	D
Def SimStatusMsg	D
Def SortMethod	D
Def TaskType	D
Def TGCode	D
Def Variable	D
Elements	X
Feeder SetAttr	D
Feeders	D
Finder Candidate Lists	D
Finder DoNotMatchAttr	D
Finder SetAttr	D
Finders	D
Generic Activities	D
Generic Tasks	D
ImportAttrXref	E
ImportUnitToSlotIDref	E
ImportValXref	E
Junc MOSID MOC	E
Junc ResClass AttrClass	E
Organization Attributes	D
Organizations	D
OUTPUT_FilteredOrgs	O
OUTPUT_FilteredRes	O
OUTPUT_FilteredSlots	O
OUTPUT_Filters	O
OUTPUT_ObjAttr	O
OUTPUT_ObjFilter	O
Output_OrgFilters	O
OUTPUT_ResourceAttributes	R
Output_ResourceAutoGens	R
Output_ResourceFilters	O
Output_Selected ResGrps	O
Output_SelectedOrgs	O
Output_SlotFilters	O
ResGrp Progress	R
ResGrp SearchPaths	D/R
ResGrp SearchResults	R
ResGrp SelectionResults	D/R
ResGrps	D
Resource Attributes	D
Resource Candidate Filters	D
Resource Candidate Lists	D

Resource Candidates	D
Resource Requirement Filters	D
Resource Requirement Lists	D
Resources	D/R
Scenario Information	D
Sender SetAttr	D
Senders	D
SetAttr Probabilities	D
Simulation Progress	R
Slot Attributes	D/R
Slots	D
SO_AttrReqList	I
SO_AttrUpdate	I
SO_CloneSlotAttrReq	I
SO_FinderActSeq	I
SO_NewResAttrRecords	I
SO_PartTypeReqs	I
SO_RawResources	I
SO_RawSlots	I
SO_ReqSlots	I
SO_ResAttrCurrent	S
SO_ResGrpCurrent	S
SO_ResourceUpdate	I
SO_SelResources	I
SO_SelResourcesSave	I
SO_SlotAttrCurrent	S
SO_WeightedSum	I
Special Organizations	D
Special Slots	D
Specific Activities	S
Specific Tasks	S
Task Activity Dependencies	D
Task Activity Finders	D
Task Activity Links	D
Task Activity Sets	D
Task Activity Timing	D
Task Generators	D
TEMP_RouteResGrptoSender	I
TG ProbOverride	D
TG Schedule	D
TGSch Spacings	D
WeightedPersSelectionModel	D
YC_ActiveActivities	I
YC_ActivityTree	I
YC_ActualActivityTimings	X
YC_Check_ResSelectionRUL	I

YC_CombinedActiveFFS_SetAttr	I
YC_CombinedFFS	I
YC_CombinedFFS_SetAttr	I
YC_FinderActSeq	I
YC_ResGrpInitSpecActivity	I
YC_RUL_ShowRemovedResources	I

Distribution list

Document No.: DRDC CORA TM 2010-261

(Report distributed by CD unless otherwise noted)

LIST PART 1: Internal Distribution by Centre

3 Authors (Hard Copies)
1 DG DRDC CORA
1 DDG DRDC CORA
1 Chief Scientist DRDC CORA
1 Section Head, Land OR
1 LFORT
1 LCDORT
2 DRDC CORA Library (1 Hard Copy, 1 CD)
1 DGMPRA Personnel Generation Research Section

12 TOTAL LIST PART 1

LIST PART 2: External Distribution by DRDKIM

1 ADM(S&T) (for distribution)
1 Director S&T Land
1 DRDKIM 3
1 DG DRDC Valcartier
1 CF College Library
1 Fort Frontenac Library
1 COS(Land Ops) [DGLS]
1 COS(Land Strat) [DGLCD]
1 LFDTS
1 DLS [DLSP]
1 G1 [DLPM]
1 G3 [DLFR]
1 G4 [DLSS]
1 DLCD
1 DLFD
1 DLR
1 DLCI
1 DAT
1 DAD
1 DLSE
1 CISTI

1 Document Exchange Manager
DSTO Research Library
Defence Science & Technology Organisation
PO Box 44
Pyrmont NSW 2009
AUSTRALIA

1 Dr. Neville J Curtis
Research Leader Land Operations Research
75 Labs
Land Operations Division
PO Box 1500
Edinburgh SA 5111
AUSTRALIA

1 Chief Analyst
Land Battlespace Systems
Dstl Integrated Systems
Room 31, Bldg A3, Fort Halstead
Sevenoaks, Kent, UK, TN14 7BP

1 Dr. Jason Field
Land Battlespace Systems
Dstl Integrated Systems
Fort Halstead
Sevenoaks, Kent, UK, TN147BP

1 Director, US AMSAA
ATTN: AMSRD-AMS-S)
392 Hopkins Road
APG, MD 21005-5071

1 Mr. Patrick O'Neill
Chief, Combat Support Analysis Division USAMSAA (ATTN: AMSRD-AMS-S)
392 Hopkins Road
APG, MD 21005-5071

1 Dr. James T. Treharne
OCA Division
Center for Army Analysis
6001 Goethals Road
Fort Belvoir, VA 22060-5230

1 Mr. Robert Barrett
Chief, International Activities
Center for Army Analysis
6001 Goethals Road
Fort Belvoir, VA 22060-5230

1 Mr. John Hughes
HQ, TRADOC Analysis Center (TRAC)
Programs & Resources Directorate (PRD)
255 Sedgwick Avenue
Fort Leavenworth, Kansas 66027-2345

1 Mr. Bob Barbier
TNO Defence, Security and Safety
Information and Operations
P.O. Box 96864, 2509 JG
The Hague, The Netherlands

31 TOTAL LIST PART 2

43 TOTAL COPIES REQUIRED

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada – CORA
101 Colonel By Drive
Ottawa, Ontario K1A 0K2

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Managed Readiness Simulator (MARS) V2: Implementation of the Managed Readiness Model

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Okazawa, S.; Ormrod, M.; Young, C.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
December 2010	76	7

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – CORA
101 Colonel By Drive
Ottawa, Ontario K1A 0K2

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC CORA TM 2010-261	

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The Managed Readiness Simulator (MARS) is a versatile program that allows the user to quickly simulate a wide range of Canadian Forces readiness Scenarios to determine if the Resources of an Establishment are able to satisfy the requirements of a set of operational Tasks. The first version of MARS (V1) was successfully applied to a preliminary analysis of the Army's plans to generate the forces required for Task Force Afghanistan. However, several aspects of the MARS V1 architecture and design were identified as factors limiting the potential of MARS to address larger and more complex scenarios. After a significant redesign, the second version of MARS (V2) now incorporates a more advanced software architecture that integrates database technology into simulation execution and a new managed readiness model with a more advanced feature set that includes support for Establishment dynamics. The purpose of this paper is to document the implementation of the MARS V2 managed readiness model in the new software architecture. The intended audience is the analyst responsible for the implementation of MARS features and capabilities. The contents are both comprehensive and detailed such that the implementation of all aspects of the model can be understood and modified if necessary.

Le programme de simulation de gestion de la disponibilité opérationnelle (programme MARS) est un programme polyvalent qui permet à l'utilisateur de rapidement simuler une vaste gamme de scénarios de disponibilité opérationnelle des Forces canadiennes afin de déterminer si les ressources d'un établissement sont en mesure de répondre aux besoins propres à un ensemble de tâches opérationnelles. La première version du programme MARS (V1) a été utilisée avec succès lors d'une analyse préliminaire des plans de l'Armée visant à mettre sur pied les forces nécessaires pour constituer la Force opérationnelle Afghanistan. Toutefois, plusieurs aspects de la conception et de l'architecture du programme MARS V1 ont été identifiés comme étant des facteurs limitant la capacité de MARS à traiter des scénarios plus importants et plus complexes. Après une restructuration en profondeur, la deuxième version de MARS (V2) incorpore désormais une architecture logicielle plus sophistiquée qui intègre une technologie de traitement de bases de données dans l'exécution de la simulation, ainsi qu'un nouveau modèle de gestion de la disponibilité opérationnelle possédant un ensemble de caractéristiques plus sophistiquées qui comprend du soutien au niveau de la dynamique de l'établissement. Le but de la présente étude est de documenter la mise en œuvre du modèle de gestion de la disponibilité opérationnelle du MARS V2 dans la nouvelle architecture logicielle. La public cible de ce document est l'analyste chargé de la mise en œuvre des caractéristiques et des capacités du programme MARS. Les points abordés sont tour à tour présentés de façon générale et en détail, de façon à ce que la mise en œuvre de tous les aspects du modèle soit bien comprise et modifiée au besoin.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Managed Readiness Simulator; MARS; MARS V2; Simulation; Readiness; Arena; Discrete Event Simulation; Simulation Runtime Database

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca